PROYECTO FINAL DE CARRERA

TÍTULO: **PROGRAMACIÓN DE UN SISTEMA EMBEDDED MULTIMEDIA**

AUTOR: **Ramón Viedma Ponce**
DIRECTOR: **Manuel Domínguez Pumar**

FECHA: **28 de Junio de 2010**

RESUMEN

Este PFC muestra como programar un sistema embedded multimedia completo. La plataforma escogida es un ARM OMAP3530 de Texas Instruments corriendo un kernel Linux como sistema operativo. El sistema consta de una placa DevKit8000 de la empresa Embest y un panel táctil de 7 pulgadas. Se explica cómo utilizar la pantalla táctil, programar una interfaz gráfica con las librerías Qt de Nokia, reproducir audio a través de la interfaz estándar en audio de Linux denominado ALSA, descodificar MP3 con la librería open-source MAD, realizar funciones de procesado en un DSP y programación de un driver completo para un conversor analógico-digital. Además se incluyen guías paso a paso de instalación y configuración de todas las herramientas y librerías necesarias, así como del proceso de compilación de algunas de ellas. Este trabajo está enfocado a la sustitución de la placa de trabajo anterior para las prácticas de laboratorio de la asignatura SETR. A partir de este trabajo, se elaborará un manual de prácticas para los futuros estudiantes de esta asignatura.

CONTENIDO

LISTA DE FIGURAS

CAPÍTULO 1
INTRODUCCIÓN

¿Qué es un sistema embedded?

Los sistemas embedded, también llamados sistemas encastrados o empotrados, son sistemas altamente integrados que ocupan poco espacio, bajo coste y tienen un consumo muy reducido. Dichos sistemas suelen estar diseñados para una aplicación muy específica, como por ejemplo, la electrónica que gobierna el funcionamiento de un enrutador (router en inglés) en redes IP.

Actualmente, los sistemas embedded juegan cada día un papel más importante en los dispositivos electrónicos que nos rodean cotidianamente. Vemos ejemplos diariamente en aparatos como televisores, reproductores de DVD, cámaras fotográficas y muchos más. Posiblemente un sistema embedded considerado de alta complejidad sea el de un teléfono móvil moderno, de los denominados *smartphones*, que integra además de las prestaciones clásicas como teléfono, una pantalla táctil, reproducción de música, conexión a Internet, grabación de vídeo y una larga lista de funciones adicionales.

La mayoría de la gente ignora que la mayoría de ordenadores que actualmente existen son precisamente sistemas embedded. De hecho, el 98%[1] de todos los dispositivos de tipo computador son sistemas embedded. El mercado de los PC, aunque a priori no lo pudiera parecer, es precisamente uno de los más minoritarios a nivel de microprocesadores. Solamente durante el año 2005 se vendieron aproximadamente 4 billones de procesadores destinados a sistemas embedded. Típicamente estos procesadores suelen venir acompañados de múltiples periféricos integrados en el propio encapsulado, con lo que se les denomina microcontroladores. Las estimaciones para los próximos años nos dicen que para este año 2010 se habrán vendido 16 billones de sistemas embedded, mientras que para el 2020 el volumen será de entorno a los 40 billones. Es por tanto un mercado en auge y en plena expansión.

Las razones para este desmesurado éxito se basan en que un sistema embedded añade mucho valor al producto al que se integra, tan alto que supera ampliamente el coste del sistema en sí. Por ello, en los próximos años se espera un incremento substancial en la incorporación de sistemas embedded en mercados tales como automoción (36%), automatización industrial (22%), telecomunicaciones (37%), electrónica de consumo (36%) y equipos de salud y medicina (33%).

[1] Fuente: Consultora Artemis en su artículo "*What is an embedded system?*", 2005

Motivación de este trabajo

Para la asignatura de SETR (Sistemes Encastats a Temps Real) de esta escuela, se ha procedido a adquirir una nueva placa de desarrollo que viene a actualizar la que se estaba utilizando en las prácticas del laboratorio hasta ahora. La antigua placa estaba basada en un ARM7TDMI, cuyo módulo de desarrollo, el NS7520 de la empresa FS Forth Systeme, incorporaba unas funciones muy limitadas con muy pocos periféricos disponibles.

La nueva placa se denomina DevKit8000 y es con mucha diferencia un sistema mucho más completo y más potente que el anterior. No obstante, el producto viene con escasa documentación en la que los alumnos de esta asignatura puedan basarse para realizar sus prácticas y poder extraer todo lo que esta placa puede ofrecer. A diferencia del completo manual del que disponen en la actualidad con la placa antigua, que fue redactado en su momento por los profesores titulares de la asignatura, no existe una guía similar para orientar al alumno.

Este trabajo final de carrera nace por tanto, bajo la necesidad de conocer y describir de forma exhaustiva todos los sistemas que integran la nueva plataforma escogida, así como sus periféricos y de cómo se programan, con la intención de poder redactar más adelante una guía de prácticas para el laboratorio de la asignatura SETR.

El trabajo empieza en el capítulo dos con una descripción de la placa Devkit8000, así como del microcontrolador principal que la gobierna, el OMAP3530 del fabricante Texas Instruments. Dicho microcontrolador tiene sus peculiaridades dada la gran cantidad de sistemas que se hayan integrados en su interior. De hecho, el Omap3 es una gama de microprocesadores de los denominados SoC (System on-a-Chip), en la que se procura que todo lo que se pueda necesitar para construir un sistema embedded quede integrado bajo el mismo encapsulado.

A continuación se explican los primeros pasos para empezar a trabajar con esta nueva placa. Esto es, conectarse a través del puerto serie para empezar a introducir los primeros comandos al sistema operativo que ya viene pre-instalado, un Embedded Linux. Luego se enseña qué herramientas es necesario instalar y la forma de emplearlas para compilar programas para la nueva plataforma. También se explica en este mismo apartado cómo está construido el sistema, qué es un gestor de arranque y cómo podemos usarlo para actualizar cosas tan vitales como el propio kernel de Linux o el sistema de ficheros rootfs.

Una vez tengamos preparado el escenario para el desarrollo, se pasa a explicar los periféricos más interesantes de cara a la programación de un sistema embedded multimedia. Dichos periféricos incluye una pantalla LCD-

TFT de 7 pulgadas de diagonal, en la que podremos mostrar gráficos e información al usuario. Dicha pantalla viene recubierta con un panel táctil de tipo resistivo, que también se analizará, pues es de vital importante que el usuario pueda seleccionar qué hacer directamente tocando la pantalla.

Otro aspecto importante será la reproducción de audio, y en concreto se explicará la plataforma de sonido estándar en Linux, denominada ALSA, y de cómo se puede aprovechar una librería Open Source para enviar audio a través de esta interfaz.

En el siguiente apartado se explicará el procesador digital de señal (DSP) que viene integrado en el propio Omap3530. Dicho DSP se suele emplear para realizar cálculos y algoritmos de tratamiento de la señal que el procesador principal haría de forma más lenta. Además de velocidad, la intención de utilizar el DSP es la de poder dejar el microprocesador principal libre para otras tareas, como por ejemplo la gestión de red, o el control del panel táctil. Se explicará como comunicarnos y programar el DSP a través de unas librerías denominadas DSPLink que la propia Texas Instruments pone a nuestra disposición gratuitamente.

En el capítulo 3 se explicarán dos aplicaciones que se han programado para este trabajo y que sirven de ejemplo de cómo utilizar todas las funciones explicadas anteriormente. La primera aplicación consiste en un cliente que se conectará a un servidor de música y podremos escuchar el audio a través del subsistema de audio integrado. Para esta aplicación se explicará el protocolo que utiliza el servidor, denominado SHOUTcast, así como la librería MAD para la descodificación de los datos en MP3.

La siguiente aplicación consiste en el desarrollo completo de un driver para esta plataforma. Concretamente se ha escogido un conversor analógico-digital ADC0831 porque aún siendo de un solo canal, utiliza un bus SPI para comunicarse con el microcontrolador para devolver el resultado de la conversión, lo que ya permite demostrar cómo debemos de comunicarnos con el kernel de Linux para poder desarrollar nuestros drivers y poder así personalizar nuestro sistema.

Por último se realiza un estudio de cómo influye la carga del procesador principal cuando éste ejecuta todas las tareas y de cómo varía cuando el DSP colabora en el trabajo de estas tareas. Esto permite evaluar la ganancia obtenida si se delega en el DSP la realización de algunos algoritmos que en un sistema embedded se realizaban tradicionalmente en el propio procesador principal.

CAPÍTULO 2

EL SISTEMA EMBEDDED MULTIMEDIA DEVKIT8000

2.1. INTRODUCCIÓN

El desarrollo de placas basadas en microprocesador que integran todo un sistema completo en unas dimensiones compactas ha resultado ser un negocio considerable para muchas empresas. Especialmente los propias fabricantes de circuitos integrados, que han visto como, ofreciendo kits de desarrollo o placas de evaluación de sus propios productos, han sido capaces de aumentar sus beneficios a la vez que ofrecen al desarrollador una plataforma sobre la que poder experimentar y desarrollar de forma rápida a un coste competitivo. Dichas placas suelen venir acompañadas de sus respectivos esquemas eléctricos, lo que facilita mucho la labor al desarrollador dado que le permite reaprovechar un diseño hardware que está comprobado y garantizado que funciona.

En el mercado podemos encontrar una gran variedad de sistemas empotrados o embedded de esta clase, algunos diseñados para un uso de propósito general, y en cambio otros con una aplicación muy específica en mente. Tal es el caso del fabricante de semiconductores Texas Instruments y de la empresa Digi-Key, el quinto distribuidor de componentes electrónicos mas grande en Norte América. Estas dos empresas crearon conjuntamente el proyecto Beagleboard, una placa de bajo coste y bajo consumo de apenas 8cm x 8cm que integra un procesador OMAP3530 de Texas Instruments y viene acompañado de diversos periféricos en la propia placa.

A continuación se hará una explicación de la placa DevKit8000 y de sus principales características como plataforma de desarrollo. También se describirá el microprocesador que incorpora así como su arquitectura y prestaciones. Más adelante se explicará cómo empezar a desarrollar para la nueva placa mediante las herramientas adecuadas y cómo comunicarnos con la placa vía puerto serie. El último apartado consiste en la explicación del gestor de arranque, pieza fundamental en un sistema embedded, que será el que nos permita actualizar el software de cualquier parte del sistema, incluido a sí mismo.

2.2. DESCRIPCIÓN DE LA PLACA DEVKIT8000

La placa nueva, desarrollada por la empresa integradora china Embest, está gobernada por el OMAP3530 (Open Multimedia Application Platform) de Texas Instruments y es una variante de la placa Beagleboard original, a la cual se le ha añadido soporte para panel táctil. Este chip está basado en un núcleo ARM Cortex-A8, un procesador digital de señal (DSP) y además incorpora múltiples periféricos integrados, tales como bus serie I2C y SPI. En la imagen siguiente, podemos ver la placa en cuestión, donde se señalan los conectores soldados en la propia placa, así como el tipo de dispositivo que podemos conectar.

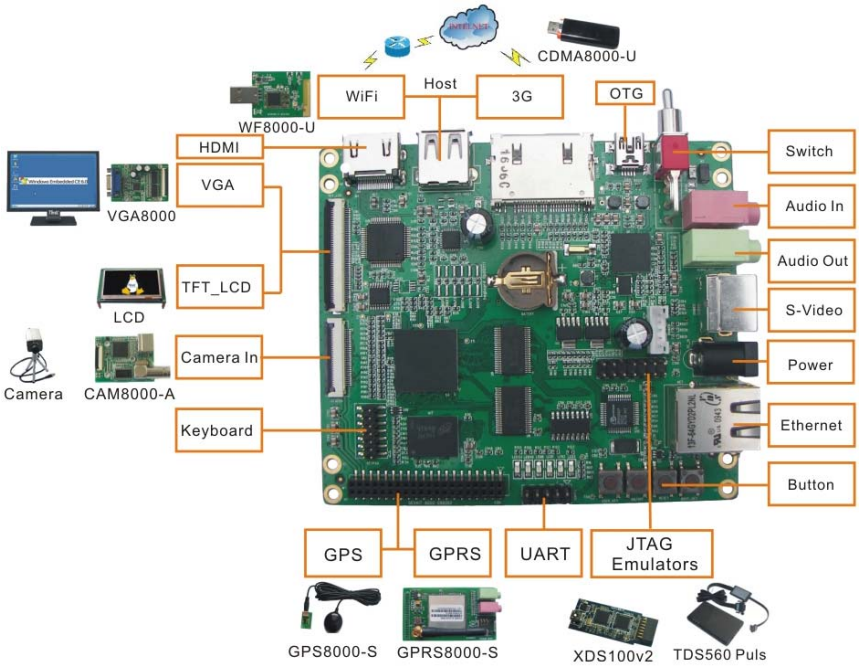


Figura 1: Aspecto de la placa Embest DevKit8000

Esta placa permite la posibilidad de conectar múltiples dispositivos gracias a los conectores estándar que vienen soldados en la placa, concretamente:

- Pantalla LCD-TFT con panel táctil de tipo resistivo
- Monitor con interfaz DVI a través del conector HDMI
- Auriculares y micrófono
- Salida de vídeo a proyector mediante conector S-Video
- Conexión a red Ethernet mediante conector RJ-45
- Módulo de cámara mediante el módulo CAM8000-A
- Conector USB tipo Host al que conectar módems o memorias
- Conector USB tipo OTG (On-The-Go), que permite a la placa ser enchufada a otro dispositivo como un PC para ser reconocido como si fuera un periférico
- Teclado de matriz a través de conector de pins
- Puerto serie RS-232 estándar, rotulado en el dibujo como UART
- Ranura para tarjetas de memoria SD
- Puerto para emulador y depurador JTAG

Además, en la placa existe un conector de 40 pins donde se han dispuesto varios puertos serie y otras señales de interés directamente del microcontrolador. Esto nos permite ampliar el número de periféricos que podemos conectar al sistema, como por ejemplo un módem de datos GPRS o un módulo de localización GPS.

El kit original de desarrollo incluía una pantalla LCD-TFT con panel táctil de 4 pulgadas, aunque finalmente para el laboratorio se han adquirido unas pantallas de dimensiones superiores, concretamente de 7 pulgadas.

DESCRIPCIÓN DEL TEXAS INSTRUMENTS OMAP3530

La gama OMAP3x de Texas Instruments representa una nueva generación de microprocesadores de los denominados SoC (System On a Chip), en los que en un mismo encapsulado no sólo encontramos el procesador principal, sino además una amplia gama de dispositivos que permite operar al integrado por sí mismo prácticamente sin necesidad de usar dispositivos externos adicionales. En la figura siguiente vemos representado el esquema interno del OMAP3530:

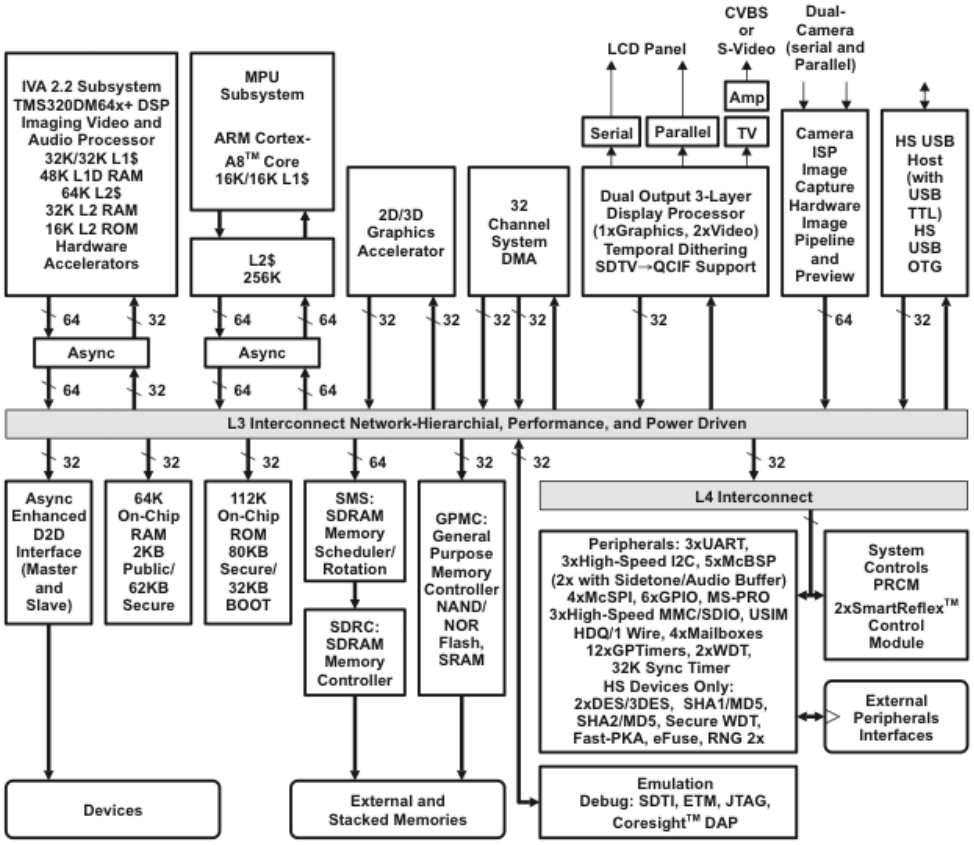


Figura 2: Arquitectura del Texas Instruments OMAP3530

Lo más destacable del esquema del OMAP3530 anterior es:

- CPU ARM Cortex-A8 a 600 MHz
- DSP TMS320DM64x+ a 425 MHz, de tipo Very Long Instruction Word (VLIW) en el subsistema de IVA (Imaging Video and Audio)
- Acelerador gráfico para 2D/3D PowerVR SGX530
- 112KB de memoria ROM y 64KB de RAM para datos y programa
- Soporte para memoria RAM externa hasta 1 GB.
- Doble salida gráfica para panel LCD o video compuesto
- Entrada para cámara de vídeo
- Subsistema Host USB para conexión de periféricos y memorias
- Múltiples periféricos: UARTs, puertos McBSP, conexión tarjetas MMC/SD, timers, bus I2C, bus SPI

Tiene por tanto, todos los elementos necesarios para poder formar un potente sistema multimedia, especialmente móvil y portátil.

2.3. PUESTA EN MARCHA INICIAL DEL SISTEMA

A continuación se describen los pasos necesarios para poner en marcha el sistema y dejarlo listo como plataforma de desarrollo. Para ello realizaremos las tareas más básicas e indispensables:

- Conectarnos al sistema mediante puerto serie RS-232
- Instalar las herramientas necesarias de desarrollo para poder compilar programas para la nueva plataforma
- Configurar un acceso NFS por red para evitar la necesidad de estar trasladando los binarios compilados en el PC de desarrollo en la placa, lo que permitirá agilizar las pruebas de los programas que vayamos desarrollando
- Explicación del gestor de arranque y las opciones que nos ofrece, concretamente la posibilidad de arrancar un kernel remoto por red, lo que nos permitirá experimentar distintos kernels que compilaremos acorde a nuestras necesidades

2.3.1. Conexión a la placa por puerto serie

Nos conectaremos a la placa a través de un puerto serie RS-232 estándar de un PC. En los PC's modernos actuales, muchos ya no incluyen ningún puerto serie. Esto es especialmente cierto en el mundo de los portátiles. Si fuera nuestro caso, podemos recurrir a un conversor USB-RS232, que es un dispositivo fácil de encontrar en tiendas del sector informático, es económico y no suele necesitar ninguna configuración especial para su instalación. Para iniciar la conexión a la placa, si estamos en un entorno Linux como Ubuntu, usaremos el *minicom*. Sobre plataformas Windows podemos usar el Hyperterminal, que viene incluido de serie en el sistema operativo. Bajo Ubuntu, si no estuviera instalado el *minicom*, lo haremos escribiendo:

```
> sudo apt-get install minicom
```

Una vez instalado, entraremos a la configuración de *minicom* tecleando:

```
> minicom –s
```

Y configuraremos el programa según los siguientes parámetros:

- Velocidad de bits y paridad: 115200 8N1
- Dispositivo serie: /dev/ttyUSB0
- Control de flujo por hardware: NO
- Control de flujo por software: NO
- Secuencia de inicialización al abrir el puerto serie: En blanco
- LineWrap (rejuntado de líneas): On

En caso de duda, en el anexo III.1 se ha explicado paso a paso con capturas de pantallas el procedimiento concreto para introducir dicha configuración.

Al finalizar el proceso de configuración, deberíamos de poder ver una pantalla del minicom como la siguiente:

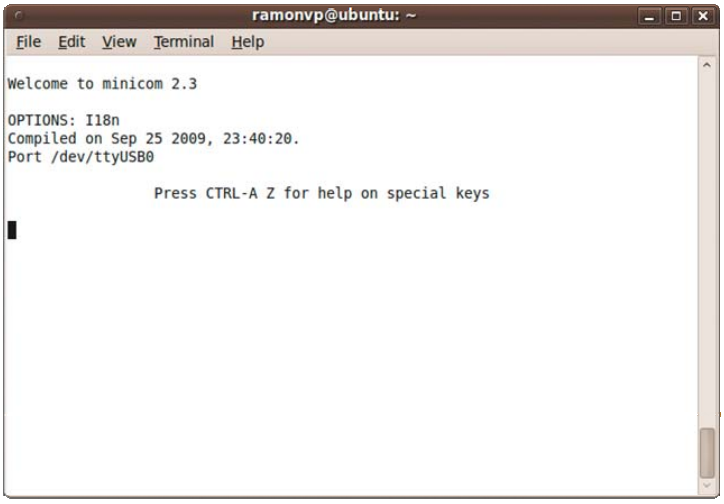


Figura 3: Pantalla inicial de minicom

Por tanto, ya tenemos todo listo para poder empezar a comunicarnos con la placa vía puerto serie. Si ahora conectamos la alimentación de la placa, veremos cómo empiezan a salir los mensajes de arranque de la placa. Más adelante se explicará la importancia de los mensajes iniciales, que corresponden con los del gestor de arranque U-Boot.

2.3.2. Instalación del toolchain

Una *toolchain*, término en inglés que significa cadena de herramientas, es un conjunto de utilidades empleadas para obtener aplicaciones para una plataforma concreta de hardware. Probablemente la *toolchain* más conocida sea la GNU toolchain, que corresponde con aquellas herramientas necesarias para crear programas en un sistema operativo Linux sobre plataforma x86. Esto comprende en su versión más simple un editor de texto, un compilador, un linker para generar los ejecutables y en un depurador para poder corregir y depurar el código. También suele formar parte del toolchain las librerías básicas contra las que compilar y enlazar los ejecutables, como la librería C estándar.

Habitualmente, cuando compilamos programas en un PC, el resultado es una aplicación que se ejecuta también en un PC. No obstante, cuando se está desarrollando un producto que tiene como objetivo un sistema embedded, la plataforma de desarrollo sigue siendo un PC, pero la máquina que ejecutará el código binario será el sistema embedded en sí. Como normalmente la

arquitectura de máquina no será la misma en el PC que en el sistema embedded, será necesario usar unas herramientas especiales que permiten compilar código para otra plataforma diferente a la plataforma sobre la que estamos compilando. Es lo que se denomina un compilador cruzado, o en inglés cross-compiler. Es decir, compilamos en un PC con arquitectura x86 (Intel, AMD), pero el resultado es un binario que se ejecutará sobre un OMAP3530, que es arquitectura ARM.

Aunque es posible descargar el código fuente de un compilador cruzado de varias fuentes de Internet y compilarlo todo para obtener una *toolchain* completa para nuestro sistema de desarrollo, suele ser más simple y más cómodo descargar algún paquete completo que incluya estas herramientas ya compiladas.

Para nuestro caso concreto, emplearemos la *toolchain* que ofrece gratuitamente la empresa CodeSourcery, que además de venir en el CD del producto, podremos descargar desde su página web la última versión.

Pasos para instalar el toolchain de Code Sourcery:

Ejecutaremos el fichero .bin que acompaña el CD en el directorio *linux/tools*. Hay que tener en cuenta que el script de instalación no soporta el shell por defecto en Ubuntu Linux, que es dash, con lo que hay que cambiar al shell bash. Para hacer este shell el de por defecto, hay que escribir lo siguiente en un terminal:

> *> sudo dpkg-reconfigure –plow dash*
> *> Install dash as /bin/sh?* ***<NO>***

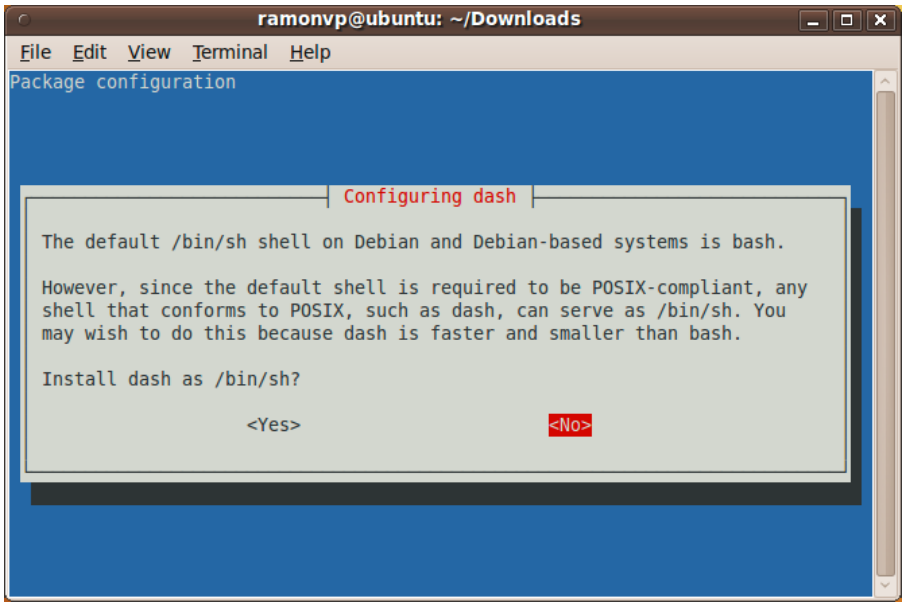


Figura 4: Cambio de shell dash a bash

Luego ya podremos instalarlo de forma habitual, ejecutando el binario:

>/bin/sh arm-2009q3-67-arm-none-linux-gnueabi.bin

Para más información acerca de la instalación del *toolchain*, se ha realizado una guía paso a paso más detallada en el anexo III.2.

Una vez instalado el *toolchain*, como paso final se procederá a obtener el código fuente de cada parte de la placa, dado que posteriormente lo necesitaremos para hacer nuestras modificaciones y compilar nuestros *kernels* personalizados. Todos los ficheros con el código fuente vienen incluidos en el CD del fabricante y se recomienda descomprimirlos ficheros en el directorio *$HOME/projects*, que será el directorio que actuará de raíz en todos los futuros desarrollos de este trabajo. Para ello, escribiremos en un terminal las siguientes instrucciones:

```
>mkdir $HOME/projects
>cd $HOME/projects
>tar xvf /media/cdrom/linux/source/x-load-1.41.tar.bz2
>tar xvf /media/cdrom/linux/source/u-boot-1.3.3.tar.bz2
>tar xvf /media/cdrom/linux/source/linux-2.6.28-omap.tar.bz2
>sudo tar xvf /media/cdrom/linux/source/rootfs.tar.bz2
```

Nótese el detalle de que para poder descomprimir el sistema de ficheros *rootfs*, es necesario tener permisos de usuario *root*, por lo que esta instrucción debe ejecutarse en modo súper usuario con el comando “*sudo*” en Linux.

2.3.3. Compartir un directorio por NFS

NFS es el acrónimo en inglés de Network File System y es un protocolo de red a nivel de aplicación según el modelo OSI diseñado para que varias máquinas puedan compartir ficheros entre sí y verlos como si estuvieran montados en el disco duro local. NFS es un protocolo bastante antiguo que ha sobrevivido el paso de los años y aún hoy sigue usándose, siendo uno de los más empleados en entornos Unix. Existen varias versiones de este protocolo, entre las que mencionamos:

- Versión 2 (NFSv2): es la más antigua y está ampliamente soportada por muchos sistemas operativos. Descrita en RFC 1094.
- Versión 3 (NFSv3): tiene más características, incluyendo manejo de archivos de tamaño variable y mejores facilidades de informes de errores, pero no es completamente compatible con los clientes NFSv2. Descrita en RFC 1813.
- Versión 4 (NFSv4): incluye seguridad Kerberos, trabaja con cortafuegos, permite ACLs (Access Control List) y utiliza operaciones con descripción del estado. Descrita en RFC 3530

PROCEDIMIENTO PARA CONFIGURAR NFS EN EL PC

Para poder habilitar un directorio compartido, primero hay que instalar el servidor NFS en el PC que actuará de host y que será donde compilaremos todos los programas. Para ello, lo instalaremos desde los repositorios estándar, escribiendo en un terminal:

```
> sudo apt-get install nfs-kernel-server
```

Una vez haya finalizado la instalación del servidor NFS, debemos de configurar la lista de directorios que se desean compartir en red. Esto se especifica en el fichero de configuración */etc/exports*, al que añadiremos con cualquier editor de texto la siguiente línea de configuración:

```
> sudo gedit /etc/exports
Añadir: $HOME/projects *(rw,sync,no_subtree_check)
```

El símbolo “*” indica que el recurso puede ser accedido desde cualquier máquina, sin importar su dirección IP o subred en la que se encuentre. El indicador “rw” declara que el que se conecte gozará de permiso para realizar lecturas y escrituras en el directorio indicado. El resto de opciones podemos consultarlas en la ayuda o en el manual y sirven para un mejor funcionamiento más fluido. Ya sólo nos quedará reiniciar el servidor:

```
> sudo /etc/init.d/nfs-kernel-server restart
```

Con el comando `sudo exportfs` podemos ver la lista de recursos compartidos y comprobar que efectivamente el directorio arriba indicado está entre los recursos compartidos.

PROCEDIMIENTO PARA CONFIGURAR NFS EN LA PLACA

Si queremos que el recurso compartido en red se monte automáticamente durante el arranque del sistema, deberemos seguir los siguientes pasos:

1. Añadir al fichero */etc/hosts* una entrada con la dirección IP del servidor para evitar escribir y recordar esta información cada vez. Con un editor de textos añadiremos la siguiente línea:

   192.168.1.47 pc

   De esta forma, cada vez que mencionemos "pc" en algún script, el sistema sabrá que hablamos de la máquina en red configurada con la dirección IP indicada.

2. Luego configuramos que monte automáticamente el directorio del pc servidor editando el fichero */etc/fstab.* Añadiremos la siguiente línea, en la que se indica qué recurso de red debe montarse en qué directorio local de nuestro sistema de ficheros:

   pc:/home/ramonvp/projects /projects defaults, nolock, rsize=8192, wsize=8192

Una vez más, es necesario especificar algunas opciones de configuración indispensables para el buen funcionamiento del protocolo de red. En este caso, se especifica el tamaño de datos de los paquetes a intercambiar mediante las opciones *rsize* y *wsize*. Esto es vital puesto que de lo contrario, el sistema se queda colgado al intentar copiar un fichero de más de 16KB a través la red.
Si ahora reiniciamos la placa con el comando "*reboot*", al finalizar el arranque ya tendremos montada la ruta anterior. Obsérvese que previamente será necesario crear el directorio */projects* en el sistema de ficheros de la placa para que posteriormente se pueda aprovechar como punto de montaje. Para crear un directorio en Linux escribiremos el comando:

```
>mkdir /projects
```

Por el contrario, si lo que queremos es montar el recurso compartido de la red a mano cuando más nos convenga, bastará con ejecutar el siguiente comando desde la línea de comandos de la placa:

```
> mount -o nolock,rsize=8192,wsize=8192 –t nfs
pc:/home/ramonvp/projects /projects
```

2.3.4. El gestor de arranque U-Boot

2.3.4.1. ¿Qué es un gestor de arranque?

Un gestor de arranque es un programa que se ejecuta al poco de arrancar la máquina. En ordenadores PC, el gestor de arranque se ejecuta cuando la BIOS (Basic Input Output System) ha terminado su inicialización. En sistemas embedded, este programa se ejecuta prácticamente tras conectar la alimentación. Este programa es muy sencillo y carece de las funcionalidades completas típicas de un sistema operativo, puesto que su misión es precisamente la de preparar todo aquello que necesita el sistema operativo para funcionar. Por ejemplo, el gestor de arranque es el que le indica al kernel de Linux qué partición montar como sistema de ficheros principal, el Root File System (rootfs).

Un gestor de arranque debe de ser capaz de cómo mínimo realizar las siguientes operaciones básicas en cualquier sistema embedded:

- Inicializar el hardware, especialmente el controlador de memoria
- Proporcionar parámetros de arranque al kernel de Linux
- Iniciar el kernel y pasarle el control

Típicamente, un gestor de arranque permite además unas operaciones muy convenientes para simplificar el desarrollo y puesta en marcha de una plataforma:

- Realizar lecturas y escrituras a posiciones de memoria arbitrarias
- Cargar nuevos ficheros binarios a la memoria RAM del sistema a través de alguna interfaz serie o incluso Ethernet
- Volcar a memoria no volátil (como tipo Flash o Nand)el contenido de la memoria RAM

U-Boot es un bootloader (gestor de arranque en inglés) libre de código abierto (open source) bajo licencia GPL orientado a sistemas embedded Linux y con soporte para una amplia variedad de plataformas. El desarrollador es libre de descargar el código fuente de su página web en Internet (http://www.denx.de/wiki/U-Boot/WebHome) para compilarlo y adaptarlo a sus necesidades y plataforma concreta.

El fabricante de la placa que nos ocupa, la DevKit8000, ya ha hecho esto por nosotros y nos entrega junto con la placa el U-Boot ya compilado y grabado en memoria listo para utilizar, además del código fuente empleado para su compilación. Para acceder al menú inicial del U-Boot cuando arranca la placa, se dispone de 3 segundos para evitar el auto-arranque. Pulsamos ENTER o cualquier otra tecla al ver el mensaje “Hit any key to stop autoboot:”.

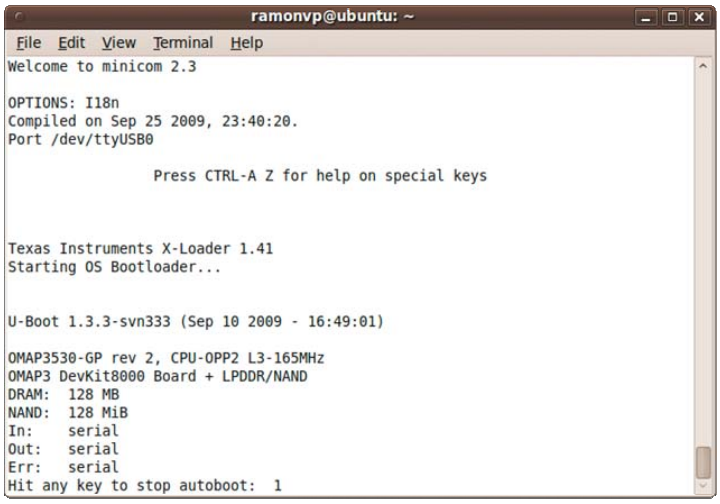


Figura 5: Acceso al gestor de arranque U-Boot

Entonces nos aparecerá el *prompt* de lo que se denomina el *monitor*:

```
> OMAP3 DevKit8000 #
```

Llegado este punto, ya podremos introducir comandos al gestor de arranque.

DIVISIÓN DE LA MEMORIA

U-Boot permite una amplia variedad de instrucciones, que podemos usar en nuestro beneficio para múltiples tareas, tales como las que se describen más adelante en los siguientes apartados. Es importante tener en cuenta que en la placa DevKit8000, cualquier operación que realicemos debe de respetar el mapa de memoria de la placa que viene pre-configurado y que es el siguiente:

Intervalo de memoria	Asignado a	Tamaño del espacio
0x00000000-0x00080000	X-Loader	512 KB
0x00080000-0x00260000	U-Boot	1.920 KB
0x00260000-0x00280000	U-Boot Env	128 KB
0x00280000-0x00680000	Kernel	4.096 KB
0x00680000-0x08000000	File System	2.041 MB

Tabla 1: Mapeado del espacio de memoria

SECUENCIA DE ARRANQUE EN LA PLACA DEVKIT8000

El OMAP3530 lleva integrada una memoria ROM mapeada en la dirección 0x00000000. El contenido de esta memoria viene pregrabado de fábrica y además es inalterable. Tras conectar la alimentación, el registro contador de programa de la CPU (PC, Program Counter) salta a esta primera dirección de memoria. El código grabado en esta memoria ROM, denominado BootROM, mira en primer lugar el estado de los pins de arranque, para determinar de qué dispositivo se debe seguir el arranque. Las opciones son múltiples y podemos arrancar incluso desde una interfaz serie, una línea ethernet, dispositivo USB, memoria externa SD o simplemente desde la memoria por defecto NAND. El fabricante ha dispuesto en la placa un botón de usuario que presionándolo al conectar la alimentación obligará a arrancar desde la tarjeta SD.

En nuestro caso, el orden habitual será seguir la ejecución desde la memoria NAND externa. Para ello, la BootROM buscará el X-Loader, que espera encontrar en la memoria NAND con un offset 0, es decir, a principio de la memoria. El X-Loader es un bootloader denominado de fase 1 y es el encargado de configurar el reloj del sistema y la memoria SDRAM. A continuación X-Loader buscará un fichero llamado “u-boot.bin”, que es el gestor de arranque U-Boot que se ha comentado anteriormente. Tras cargarlo en la memoria RAM, el X-Loader realiza un salto a esta dirección de memoria, con lo que se empieza a ejecutar el gestor de arranque U-Boot.

OPERACIONES QUE PODEMOS HACER DESDE U-BOOT

A continuación se describen algunas de las operaciones que vamos a poder realizar desde el propio gestor de arranque y que son esenciales para la administración y configuración del sistema. Todas estas operaciones tienen en común que se debe de grabar el fichero que se quiere actualizar en una memoria SD. Una vez insertada la memoria en el zócalo de la placa, se procederá a conectar la alimentación y, como se ha explicado en el punto anterior, accederemos al *prompt* del bootloader U-Boot. Una vez hecho esto, se escribirán los comandos necesarios para actualizar el software que nos interese según las siguientes secciones.

2.3.4.2. ACTUALIZACIÓN DEL X-LOADER

Para actualizar el X-Loader, escribiremos los siguientes comandos en U-Boot:

```
OMAP3 DevKit8000# mmcinit
OMAP3 DevKit8000# fatload mmc 0:1 80000000 x-load.bin.ift_for_NAND
reading x-load.bin.ift_for_NAND
9664 bytes read
OMAP3 DevKit8000 # nand unlock
device 0 whole chip
nand_unlock: start: 00000000, length: 134217728!
NAND flash successfully unlocked
OMAP3 DevKit8000 # nand ecc hw
OMAP3 DevKit8000 # nand erase 0 80000
NAND erase: device 0 offset 0x0, size 0x80000
Erasing at 0x60000 -- 100% complete.
OK
OMAP3 DevKit8000 # nand write.i 80000000 0 80000
NAND write: device 0 offset 0x0, size 0x80000
Writing data at 0x7f800 -- 100% complete.
524288 bytes written: OK
```

2.3.4.3. ACTUALIZACIÓN DEL U-BOOT

Actualizar el gestor de arranque es importante, puesto que alguna característica que podamos necesitar puede no estar disponible en la versión del fabricante, por lo que tendremos que compilar y cargar una versión más actual de este software. Para ello, escribiremos los siguientes comandos, también en U-Boot:

```
OMAP3DevKit8000# mmcinit
OMAP3 DevKit8000 #      fatload mmc 0:1 80000000 flash-uboot.bin
reading flash-uboot.bin
1085536 bytes read
OMAP3 DevKit8000 #      nand unlock
device 0 whole chip
nand_unlock: start: 00000000, length: 134217728!
NAND flash successfully unlocked
OMAP3 DevKit8000 #      nand ecc sw
OMAP3 DevKit8000 #      nand erase 80000 160000
NAND erase: device 0 offset 0x80000, size 0x160000
Erasing at 0x1c0000 -- 100% complete.
OK
OMAP3 DevKit8000 #      nand write.i 80000000 80000 160000
NAND write: device 0 offset 0x80000, size 0x160000
Writing data at 0x1df800 -- 100% complete.
1441792 bytes written: OK
```

2.3.4.4. ACTUALIZACIÓN DEL KERNEL

Necesitamos poder actualizar el kernel de forma habitual, puesto que cada elemento hardware que deseemos incorporar o probar en nuestro sistema, necesitará de su correspondiente driver. Normalmente, el driver o controlador, es preferible integrarlo en el kernel y que ya esté cargado y disponible en el momento de arrancar la placa.

```
OMAP3 DevKit8000 # mmcinit
OMAP3 DevKit8000 # fatload mmc 0:1 80000000 uImage
reading uImage
1991900 bytes read
OMAP3 DevKit8000 # nand unlock
device 0 whole chip
nand_unlock: start: 00000000, length: 268435456!
NAND flash successfully unlocked
OMAP3 DevKit8000 # nand ecc sw
OMAP3 DevKit8000 # nand erase 280000 200000
NAND erase: device 0 offset 0x280000, size 0x200000
Erasing at 0x460000 -- 100% complete.
OK
OMAP3 DevKit8000 # nand write.i 80000000 280000 200000
NAND write: device 0 offset 0x280000, size 0x200000
Writing data at 0x47f800 -- 100% complete.
2097152 bytes written: OK
```

2.3.4.5. ACTUALIZACIÓN DEL SISTEMA DE FICHEROS (ROOTFS)

El sistema de ficheros o *rootfs* (Root File System) contiene todos los programas ejecutables y ficheros de datos que nuestra aplicación va a necesitar, incluyendo utilidades clásicas de un sistema Linux tradicional. Así pues para empezar, necesitaremos incorporar librerías y ejecutables al *rootfs* para poder probar y ejecutar las aplicaciones que vayamos desarrollando.

```
OMAP3 DevKit8000 # mmcinit
OMAP3 DevKit8000 # fatload mmc 0:1 80000000 ubi.img
reading ubi.img
12845056 bytes read
OMAP3 DevKit8000 # nand unlock
device 0 whole chip
nand_unlock: start: 00000000, length: 268435456!
NAND flash successfully unlocked
OMAP3 DevKit8000 # nand ecc sw
OMAP3 DevKit8000 # nand erase 680000 7980000
NAND erase: device 0 offset 0x680000, size 0x7980000
Erasing at 0x7fe0000 -- 100% complete.
OK
OMAP3 DevKit8000 # nand write.i 80000000 680000 $(filesize)
NAND write: device 0 offset 0x680000, size 0xc40000
Writing data at 0x12bf800 -- 100% complete.
12845056 bytes written: OK
```

2.3.4.6. ARRANCAR UN KERNEL REMOTO POR TFTP

Es posible configurar la placa para que arranque desde un kernel ubicado en una máquina remota, en vez de usar el kernel grabado en su memoria NAND. Para ello, vamos a seguir los siguientes pasos:

1- Instalar y configurar el servidor de TFTP en el host con Ubuntu

```
> sudo apt-get install tftpd-hpa
> sudo nano /etc/default/tftpd-hpa
```

En este fichero modificaremos la siguiente línea:
RUN_DAEMON="yes"

Arrancamos el servidor con el comando:

```
> sudo /etc/init.d/tftpd-hpa start
```

Podemos comprobar que está ejecutándose:

```
> ps aux | grep tftpd
root     32503  0.0  0.0   2200   284 ?        Ss   07:50   0:00
/usr/sbin/in.tftpd -l -s /var/lib/tftpboot
```

También podemos comprobar que efectivamente el servidor está a la escucha:

```
> sudo netstat -plun | grep tftp
udp   0      0 0.0.0.0:69              0.0.0.0:*   32503/in.tftpd
udp6  0      0 :::69                   :::*        32503/in.tftpd
```

Copiamos un kernel de ejemplo para luego comprobar que funciona el arranque remoto:

```
> sudo cp $HOME/projects/linux-2.6.28-omap/arch/arm/boot/uImage
/var/lib/tftpboot
```

Arranque desde la placa DevKit8000
Cuando arranque la placa, paramos el auto-arranque presionando cualquier tecla. Entonces tenemos que configurar la dirección IP del servidor al que queremos conectarnos y la nuestra propia. Para ello configuraremos unas variables de entorno globales del propio U-Boot:

```
# setenv serverip 192.168.1.47 (dirección IP del PC servidor)
# setenv ipaddr 192.168.1.48   (dirección IP de la placa DevKit8000)
# setenv netmask 255.255.255.0 (máscara de subred)
```

Una vez configurado estos datos, ya podemos descargar la imagen del kernel y copiarla en la memoria RAM. Para ello ejecutaremos el comando *tftpboot*. La forma de usar este comando es:

tftpboot [loadAddress] [[hostIPaddr:]bootfilename]

Con lo que ejecutaremos:

```
OMAP3 DevKit8000 # tftpboot 80300000 uImage
```

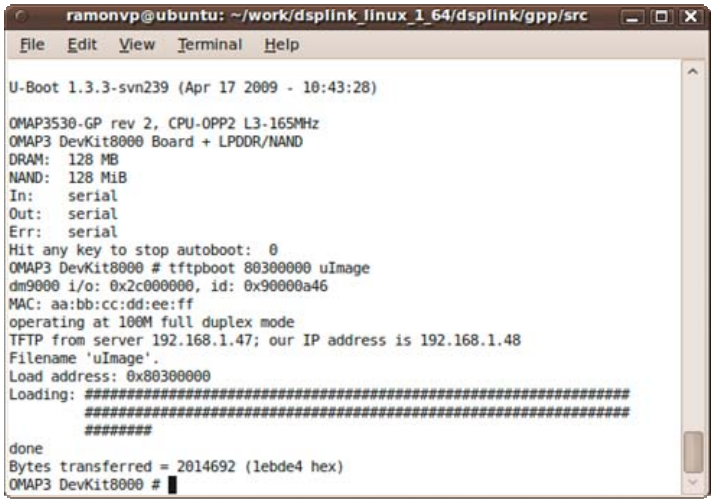


Figura 6: Carga de un kernel remoto a través de TFTP

Veremos entonces cómo va leyendo a través de la red el fichero indicado y lo va copiando a memoria. Por último, le indicamos al U-Boot que deseamos pasar el control a la dirección de memoria donde hemos copiado el kernel, para lo cual:

```
OMAP3 DevKit8000 #  bootm 80300000
```

Este método permite pues hacer muchas pruebas con configuraciones de distintos kernels, pero con la ventaja de no tener que estar regrabando cada vez la imagen del kernel de prueba en la tarjeta SD.

2.4. DISPOSITIVOS PERIFÉRICOS DISPONIBLES

El OMAP3530 es un chip de tipo SoC (System-on-a-Chip), en el que se ha integrado, además del microprocesador, unos dispositivos periféricos adicionales. Estos periféricos se podrían considerar básicos para el desarrollo de un ordenador completo. En este apartado se comentan tres de estos dispositivos por considerarse clave en el desarrollo de un sistema multimedia que permita la interacción de forma gráfica con personas. En concreto, se analizarán:

1. **El panel LCD y el Frame Buffer**: Estos elementos conforman el subsistema gráfico de salida del sistema, por lo que resultan absolutamente esencial en un dispositivo multimedia. Se explicará su configuración y algunas herramientas para acceder al Frame Buffer y poder realizar capturas de pantalla.
2. **El panel táctil**: Un elemento clave, que cada vez tiene más importancia en un sistema multimedia: la capacidad de interactuar con el sistema de forma táctil, es decir, tocando sobre la pantalla directamente las opciones que se nos presentan. En concreto, se explicará la tecnología asociada, para luego dar paso al controlador que calcula la coordenada pinchada por el usuario. Como nexo de unión entre el apartado anterior de gráficos LCD y el sistema táctil, se explican las librerías Qt de Nokia y se muestra como utilizarlas para crear un entorno gráfico basado en elementos táctiles.
3. **El subsistema de audio**: En este caso, el subsistema de audio se implementa en un circuito integrado externo, pero que al ir tan íntimamente ligado, resulta más conveniente analizarlo en esta sección. Aquí se explicará cómo compilar y utilizar las librerías de usuario ALSA (Advanced Linux Sound Architecture), que es la plataforma estándar para audio en Linux.

2.4.1. El panel LCD y el Frame Buffer

El OMAP3530 cuenta con un subsistema gráfico completo y su misión es la de proveer la lógica necesaria para mostrar una imagen de video almacenada en una memoria SDRAM o SRAM en una pantalla de cristal líquido (LCD) o bien en un televisor estándar. El sistema ofrece las siguientes características gráficas:

- Profundidad de color programable: 1, 2, 4, 8, 12, 16, y 24 bits por píxel
- Resoluciones soportadas:
    - XGA - 1024x768
    - WXGA - 1280x800
    - SXGA+ - 1400x1050
    - HD 720p - 1280x720
- 256 x 24-bit entradas de paleta en rojo, verde y azul (RGB)
- Frecuencia de píxel programable hasta 75 MHz

En el esquema siguiente, se muestra el subsistema gráfico completo:

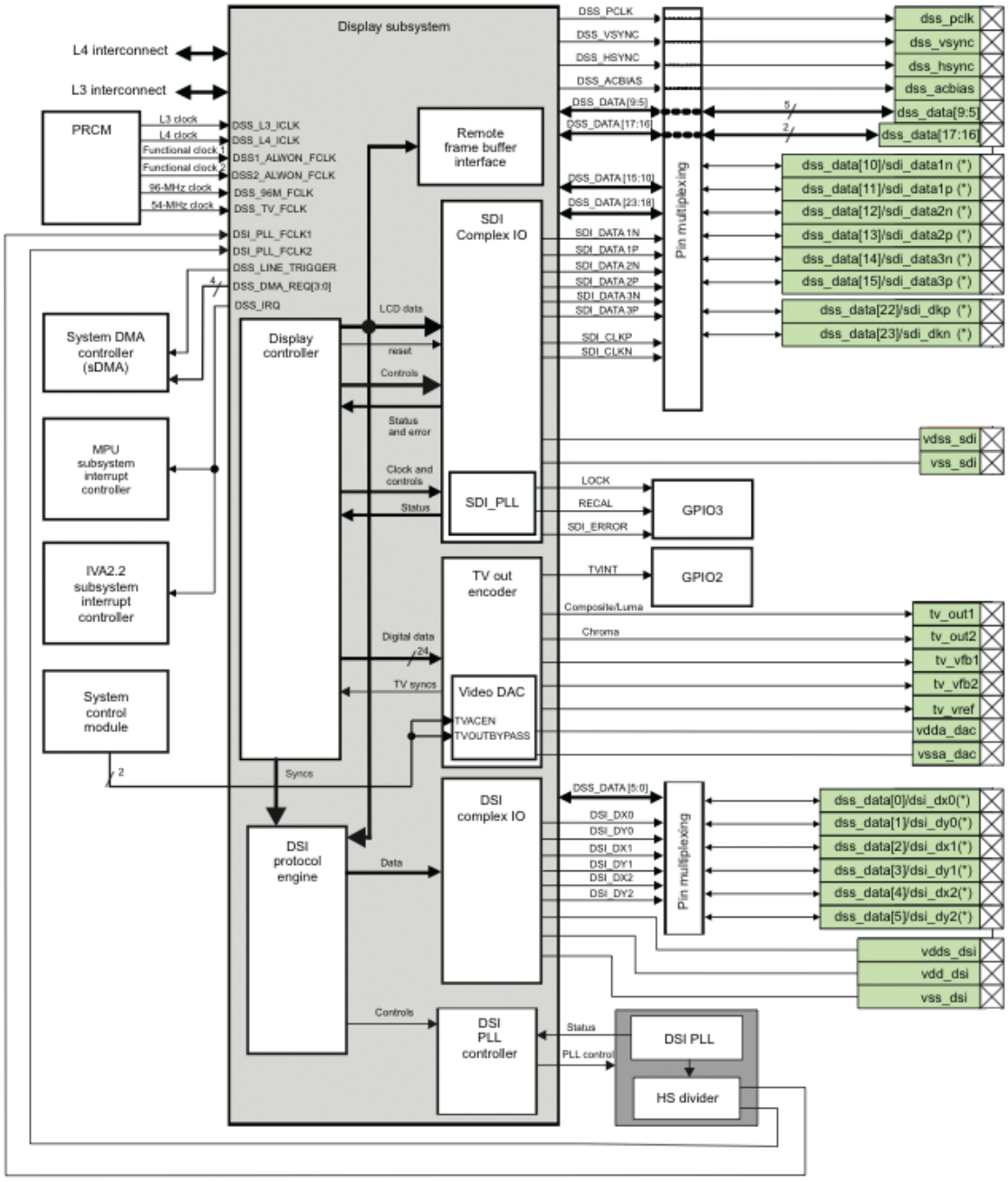


Figura 7: Subsistema gráfico del OMAP3530

En la placa DevKit8000 se ha añadido además un conversor de vídeo Texas Instruments TFP410, cuya función es la de convertir la señal de vídeo que recibe el LCD (líneas dss_data[…]) a una interfaz DVI (Digital Video Interface). De esta forma podemos conectar la placa directamente a cualquier monitor de ordenador con este tipo de conector. No obstante, el formato físico del conector es del tipo HDMI, puesto que DVI es parcialmente compatible con el estándar HDMI (High Definition Multimedia Interface) en su versión digital DVI-D y compatible también con el estándar VGA en su versión analógica DVI-A.

Figura 8: Detalle del cable conversor HDMI-DVI

Un *frame buffer* es una capa de abstracción entre el software y el hardware gráfico. Esto quiere decir que las aplicaciones no necesitan saber qué tipo de hardware se encuentra disponible en el sistema, sino que únicamente tienen que saber cómo comunicarse con éste a través de la API (Application Programming Interface) del controlador que ofrece el sistema operativo. Dicha API está bien definida y estandarizada.

Activando la pantalla LCD

La placa de desarrollo DevKit8000 se puede adquirir conjuntamente con una pantalla LCD con panel táctil integrado. En el momento de la compra, se puede decidir si queremos una pantalla de 4'' o bien de 7''. En nuestro caso, se eligió la opción de la pantalla de 7''. No obstante, si conectamos la pantalla a la placa y conectamos la alimentación, veremos que no aparece nada de especial en pantalla. Esto es porque se necesita realizar una primera configuración inicial desde el gestor de arranque, que se detalla a continuación:

1. Acceder al intérprete de comandos del gestor de arranque (consultar el apartado 2.3.4 para más detalles)
2. Escribiremos la siguiente línea[2] seguida de un *ENTER* cuando nos haya aparecido el *prompt*:

```
setenv bootargs console=ttyS2,115200n8 ubi.mtd=4 root=ubi0:rootfs
rootfstype=ubifs video=omapfb:mode:7inch_LCD
```

3. Vemos en letra negrita lo nuevo que se ha tenido que añadir a la variable de entorno *bootargs* del U-Boot, que almacena los parámetros

[2] *Nota: En el manual se indica que además debe ejecutarse la siguiente orden:*

setenv bootcmd nand read.i 80300000 280000 ***200000****; bootm 80300000*

El valor que aparece en negrita, ***200000****, provoca un error de Checksum del kernel al arrancar la placa. Por lo tanto, no debe de modificarse dicha configuración. Si ya se hubiera hecho, se debe reescribir el comando pero sustituyendo el* ***20000*** *por* ***21000****.*

que le pasará al kernel en el momento de su arranque. En concreto, esta línea le está indicando al kernel que existe un dispositivo de video de tipo *omap frame buffer*, en su variante de LCD de 7 pulgadas.

4. Por último, guardaremos los cambios con el comando *saveenv* y realizaremos un reset del sistema, bien sea escribiendo la orden *reset* o bien desconectando la alimentación directamente y volviéndola a conectar.
   Cuando hayamos reseteado la placa, y tras finalizar la carga del sistema operativo, veremos lo siguiente en la pantalla LCD:

Figura 9: Pantalla inicial de Linux en el LCD

Como puede apreciarse en la esquina superior izquierda de la imagen, aparece el logotipo de Linux, el pingüino Tux, que es su mascota oficial, elegida por el propio Linus Torvalds, creador de Linux. Si se desea sustituir este logo por algún otro personalizado, se debe hacer desde la configuración del propio kernel, puesto que va integrado en el propio código binario final.

La imagen que se desee colocar como logotipo inicial debe estar en un formato de fichero muy concreto, no siendo válido ningún formato habitual como JPEG, GIF, BMP o PNG. Para ello, existen algunas utilidades que permiten realizar esta conversión de una forma más sencilla. A continuación se explica uno de estos paquetes, en concreto FBLOGO.

FBLOGO – Sustituir el logo de arranque del sistema

En primer lugar, será necesario descargar el paquete FBLOGO desde Internet. Podremos hacerlo por ejemplo desde la siguiente dirección:

http://packages.debian.org/etch/i386/fblogo/download

También necesitaremos el paquete Netpbm, que es un paquete para manipular imágenes gráficas, y donde va incluido el binario pngtopnm, para lo cual escribiremos en nuestro sistema Linux Ubuntu:

```
>sudo apt-get install netpbm
```

Podemos encontrar más información de este paquete en su página web oficial:

http://netpbm.sourceforge.net/

A continuación crearemos la imagen que deseamos establecer como logo y la guardamos en formato PNG. Las restricciones imponen que la imagen tiene que limitarse a 224 colores y un máximo de 80x80 píxeles.

Una vez hecho esto, convertiremos la imagen a un formato adecuado para insertar en el kernel. Para lo cual ejecutaremos el comando:

```
>pngtopnm logo.png | pnmtoplainpnm > logo_linux_clut224.ppm
```

Para poderlo compilar con el propio kernel, se debe de mover el fichero resultante al árbol de directorios del kernel, en concreto al directorio:

```
>cp logo_linux_clut224.ppm /usr/src/linux/drivers/video/logo/
```

Es posible que en el momento de la compilación del kernel aparezca un mensaje de aviso diciendo "*Image has more than 224 colors*", debido a las limitaciones sobre la imagen original anteriormente explicadas. Para arreglarlo se puede intentar:

```
>pngtopnm logo.png | ppmquant -fs 223 | pnmtoplainpnm >
logo_linux_clut224.ppm
```

Esto reducirá el número de colores en la paleta del fichero final resultante. Tras esto, ya podemos recompilar el kernel y actualizarlo en la placa según el procedimiento explicado en el apartado 2.3.4.4.

PSPLASH – Progreso durante el arranque de Linux

Habitualmente, desde el momento en el que conectamos la alimentación de un sistema hasta que finalmente está totalmente cargado y operativo, suele pasar un intervalo de tiempo que puede llegar a ser de algunas decenas de segundos e inferior a 1 minuto. Durante este tiempo, la placa Devkit8000 no muestra ninguna información particular en pantalla, con lo que a algunos usuarios puede resultarles molesto y confuso no saber qué está sucediendo con el sistema durante este lapso de tiempo.

Es por esto que muchas empresas instalan en el arranque de sus sistemas un indicador de progreso, que típicamente suele representarse con un fondo gráfico y una barra que va llenándose a medida que la carga del sistema avanza. Incluso es posible escribir mensajes de estado, indicando qué fase está atravesando el proceso de carga.

Existen varios paquetes que realizan esta función de forma amigable para el ensamblador de sistemas. Uno de estos paquetes es el PSPLASH y que pasamos a comentar en esta sección. El código fuente de esta utilidad puede descargarse gratuitamente desde su página web:

http://labs.o-hand.com/psplash/

Para poder compilar este paquete necesitamos saber el nombre de nuestra arquitectura PC. Para ello, necesitamos un script común en muchas instalaciones denominado *'config.guess'* y *'config.sub'*, y que podemos descargar de:

```
> wget 'http://savannah.gnu.org/cgi-
bin/viewcvs/*checkout*/config/config/config.guess'
> wget 'http://savannah.gnu.org/cgi-
bin/viewcvs/*checkout*/config/config/config.sub'
```

Ejecutando el script anterior:

```
> sh config.guess
```

Se obtiene el resultado, que en la máquina de pruebas es: **i686-pc-linux-gnu**
Con este dato, procedemos a la compilación de Psplash, para lo que teclearemos en una terminal lo siguiente:

```
> ./configure --build=i686-pc-linux-gnu --host=arm-none-linux-
gnueabi
> make
```

Al compilar, se crean realmente dos ejecutables. El primero es *psplash* y actúa de servidor. Al cargarlo, aparece en pantalla una imagen con la barra de relleno vacía y sin ningún mensaje. El segundo ejecutable que se compila es *psplash-write*, que actúa de cliente y es el que le indica al servidor *psplash* qué debe de mostrar en pantalla. Para ello, existen 3 mensajes básicos que controlan el estado de la barra de progreso y el mensaje que muestra, que son los siguientes:

```
> psplash-write "MSG Cargando...23%"
> psplash-write "PROGRESS 23"
> .....
> psplash-write "QUIT"
```

El parámetro MSG indica qué texto debe mostrarse encima de la barra de progreso. El comando PROGRESS establece qué tanto por ciento debe rellenarse la barra de progreso. Por último, el comando QUIT cierra el servidor *psplash*. Con los comandos anteriores, *psplash* se ve así:

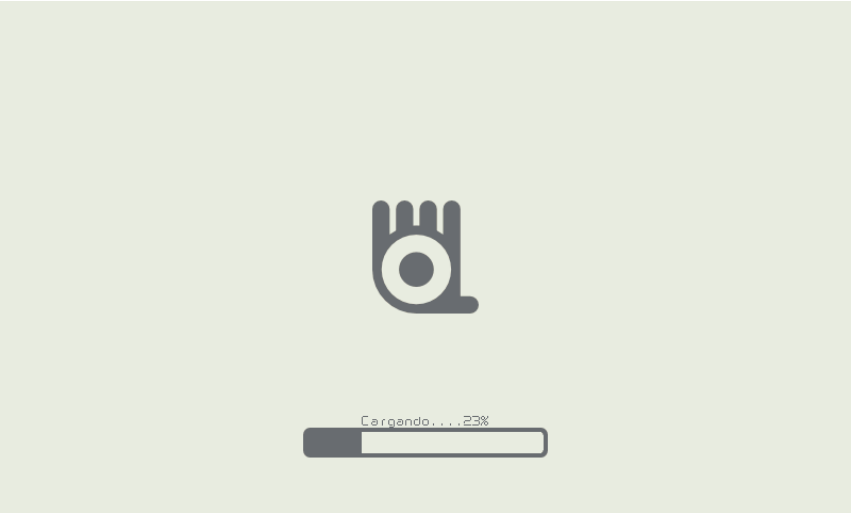


Figura 10: Pantalla de progreso con Psplash

Para poder integrar nuestras propias imágenes, necesitaremos poder convertir los .png a .h, para ello necesitamos instalar la siguiente librería:

```
> sudo apt-get install libgtk2.0-dev
```

Hecho esto, nos bastará con ejecutar el script que acompaña al código fuente del programa para obtener la imagen en formato .h para poder integrar en el programa ejecutable:

```
>./make-image-header.sh logo.png HAND
```

Esto producirá el fichero: *logo-img.h*. Habrá entonces que renombrarlo a *psplash-hand-img.h* para que al compilar con *make* quede integrado junto con el programa:

```
> mv logo-img.h psplash-hand-img.h
> make
```

FBSHOT - Capturas de pantalla del LCD

Con frecuencia nos interesará realizar una captura de lo que vemos en la pantalla LCD. Esto nos ayudará a generar documentación de ayuda paso a paso para un posible manual, o obtener imágenes de demostración del producto con fines publicitarios. Existen varios paquetes de software que son compatibles con el Linux Frame Buffer y que permiten salvar en un fichero PNG

o BMP el contenido de esta memoria gráfica. Uno de los ellos, el FBSHOT, se explica en esta sección.

Para empezar, descargaremos el código fuente de:

http://www.sfires.net/fbshot/
fbshot-0.3.tar.gz (16.2 KB)

Para poder compilar esta utilidad, necesitamos primero instalar y compilar dos librerías importantes:

- **libpng**: para manipular ficheros gráficos en formato PNG. Podemos descargar el código fuente de:

http://prdownloads.sourceforge.net/libpng/libpng-1.4.0.tar.gz?download

- **zlib**: librería de funciones para compresión. La descargaremos de:

http://www.zlib.net/zlib-1.2.3.tar.gz

Como primer paso, procederemos a compilar estas dos librerías mencionadas. Los pasos para compilarlas son idénticos para ambas, con lo que se muestra como compilar *zlib* a modo de ejemplo.

Conviene verificar que hemos creado la variable de entorno TOOLCHAIN, que apunta al directorio donde tenemos instaladas las herramientas de compilación.

Compilación de ZLIB

Ejecutaremos en primer lugar el script de configuración, necesario para establecer variables de entorno que necesitará *make* para compilar correctamente el paquete:

```
>./configure --prefix=$TOOLCHAIN
```

Antes de compilar, hay que modificar el fichero Makefile y poner los arm-none-linux-gnueabi por delante de todas las herramientas de compilación veamos como gcc, ld, ranlib, etc.... Grabamos los cambios y luego ejecutamos:

```
> make
> make install
```

Una vez compilado, nos aseguraremos de que las librerías (ficheros .a y .so) y que sus correspondientes cabeceras (ficheros .h) se encuentren en el directorio */lib* y */include* del toolchain respectivamente, lo que permitirá compilar otros paquetes que hacen uso de esta librería sin tener que volver a compilar.:

```
cp zlib.h zconf.h compiled/arm-none-linux-gnueabi/include
chmod 644 compiled/arm-none-linux-gnueabi/include/zlib.h
compiled/arm-none-linux-gnueabi/include/zconf.h
cp libz.a compiled/arm-none-linux-gnueabi/lib
cd compiled/arm-none-linux-gnueabi/lib; chmod 755 libz.a
cd compiled/arm-none-linux-gnueabi/lib; if test -f libz.so.1.2.3;
then \
        rm -f libz.so libz.so.1; \
        ln -s libz.so.1.2.3 libz.so; \
        ln -s libz.so.1.2.3 libz.so.1; \
        (ldconfig || true)  >/dev/null 2>&1; \
      fi
cp zlib.3 compiled/arm-none-linux-gnueabi/share/man/man3
chmod 644 compiled/arm-none-linux-gnueabi/share/man/man3/zlib.3
```

Compilaremos a continuación de forma similar la librería lbpng. Para compilar finalmente la utilidad FBSHOT, seguiremos los mismos pasos que para otras utilidades anteriores:

```
> ./configure --build=i686-pc-linux-gnu --host=arm-none-linux-
gnueabi
> make
```

Por último copiaremos el fichero binario *fbshot* a un directorio de la placa desde el que podamos llamarlo cuando lo necesitemos. Un buen lugar sería el directorio general de */bin*.

2.4.2. El panel táctil y su controlador

Las pantallas táctiles (touch screens en inglés) se han popularizado mucho en los últimos años, especialmente en algunos sectores como los del automóvil o en la telefonía móvil. En vehículos de alta gama y en muchos teléfonos inteligentes, denominados smartphones, una pantalla táctil es el elemento diferenciador respecto a otros productos similares de la competencia.

La posibilidad de interactuar de forma táctil con un sistema electrónico introduce en la naturaleza de su uso otra dimensión distinta a la interfaz basada en botones mecánicos tradicionales. Una pantalla táctil crea una experiencia de usuario más rica, más íntima y personal que es difícil de lograr con botones clásicos.

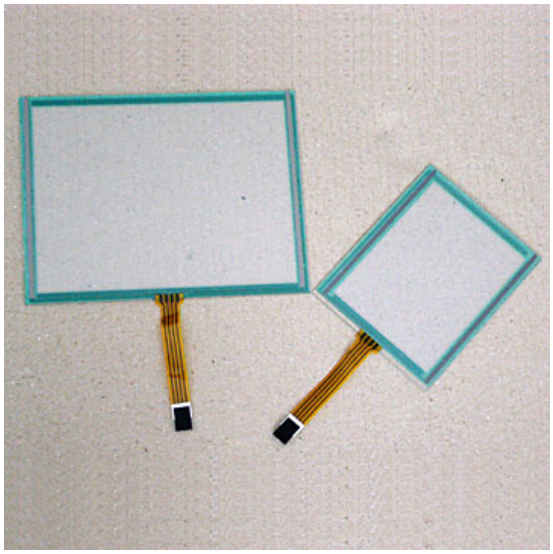


Figura 11: Paneles táctiles resistivos de 4 contactos

Debido a la fabricación a gran escala, el precio de estos dispositivos ha bajado significativamente, lo que ha permitido su introducción en muchos más dispositivos que ya no consideraríamos artículos de lujo o alta gama. Ejemplos de ello los encontramos en frigoríficos inteligentes, microondas y otros electrodomésticos, cámaras fotográficas digitales, impresoras, entre otros.

Tecnología de un panel táctil

En la actualidad existen diversas tecnologías para la fabricación de paneles táctiles. Los más populares y extendidos son:

- Resistivos: basados en medir resistencias eléctricas
- Capacitivos: basados en medir distribuciones de carga eléctrica
- Ondas acústicas superficiales (SAW: Surface Acoustic Waves): se construyen mediante emisores de ultrasonidos
- Infrarrojos: una matriz de emisores y receptores de luz permiten saber en qué punto se ha pinchado al interrumpir la luz en esa fila y esa columna

En nuestra placa Devkit8000, el tipo de panel existente es del tipo resistivo, por lo que a continuación explicaremos sus fundamentos un poco más en profundidad.

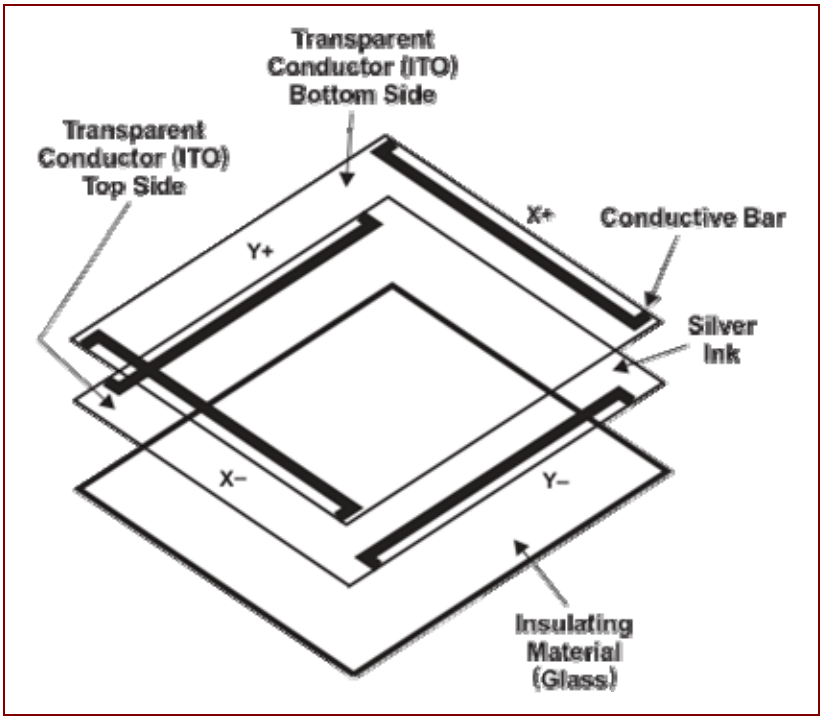


Figura 12: Estructura de un panel táctil resistivo

Una pantalla táctil resistiva está formada por varias capas. Las más importantes son dos finas capas de material conductor transparente entre las cuales hay una pequeña separación. Cada una de estas capas dispone de unos electrodos en sus laterales, que son las partes que irán soldadas con cables para poder conectar a la placa. De aquí sale el nombre de pantalla táctil a 4 hilos (4-wire en inglés). En la actualidad existen pantallas táctiles a 4, 5, 7 y 8 hilos, siendo las de 4 hilos las más populares fundamentalmente por su coste económico.

Las pantallas táctiles resistivas son por norma general más accesibles pero tienen una pérdida de aproximadamente el 25% del brillo debido a las múltiples capas necesarias. Otro inconveniente que tienen es que pueden ser dañadas por objetos afilados. Por el contrario no se ven afectadas por elementos externos como polvo o agua, razón por la que son el tipo de pantallas táctiles más usado en la actualidad

Cuando algún objeto toca la superficie de la capa exterior, las dos capas conductoras entran en contacto en un punto concreto. De esta forma se produce un cambio en la corriente eléctrica que permite a un controlador calcular la posición del punto en el que se ha tocado la pantalla midiendo la

resistencia. Algunas pantallas pueden medir, aparte de las coordenadas del contacto, la presión que se ha ejercido sobre la misma.

El controlador TSC2046 : Calculando las coordenadas

El funcionamiento más preciso es el siguiente: un controlador aplicará una diferencia de potencial en los dos extremos de una de las capas mientras que la otra capa actuará como si se tratase de una resistencia variable, así, en función de la presión que se ejerza y de donde se efectúe ésta, el potencial detectado por el controlador variará proporcionando un valor binario que representa, por ejemplo, la coordenada X. A continuación se hace lo mismo pero con la otra capa, es decir se aplica potencial a la otra capa y la que queda actúa de resistencia, obtenemos así la coordenada Y. Este proceso se repite a gran velocidad y es posible tomar del orden de 200 o más muestras por segundo.

Esto es precisamente el trabajo que realiza un controlador de panel táctil resistivo, como es el caso del TSC2046 que viene integrado en la placa DevKit8000. Un controlador para panel resistivo no es más que un circuito que integra un conversor analógico-digital (ADC) de alta velocidad junto con una electrónica que permite conmutar entre 4 líneas, tanto para tomar las medidas como para fijar tensiones en dichos terminales. A continuación puede verse el diagrama de bloques del TSC2046.

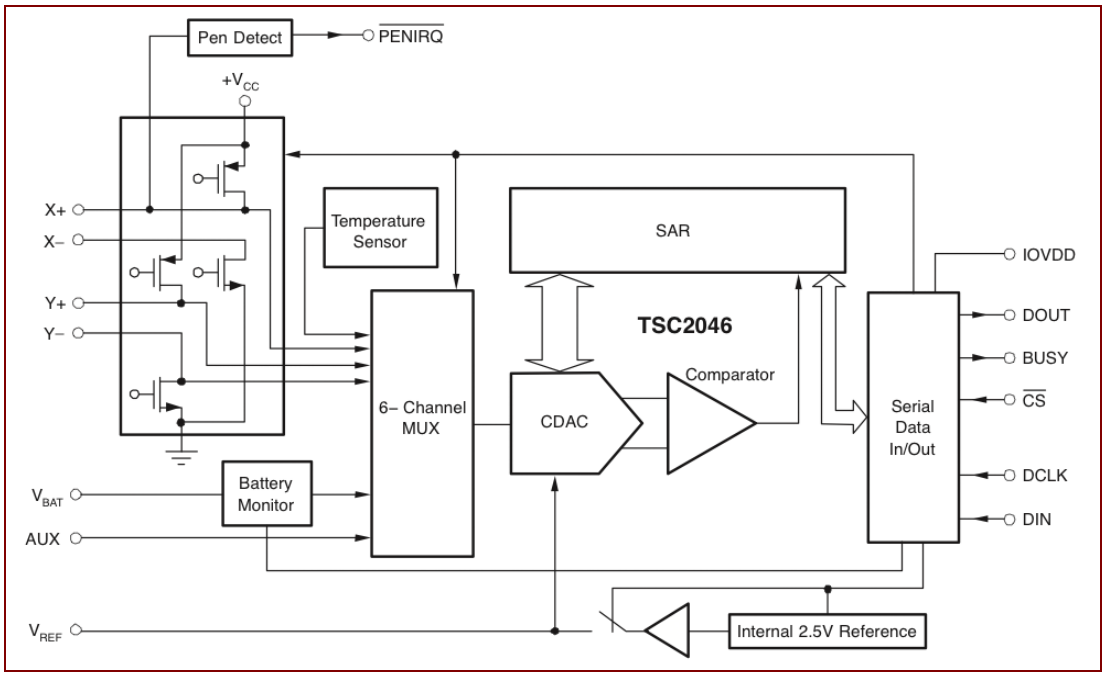


Figura 13: Esquema de bloques del controlador TSC2046

Cuando el usuario presiona el panel y pone en contacto las dos caras conductoras, se está formando el siguiente circuito equivalente:

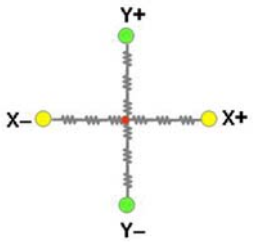


Figura 14: Divisor resistivo equivalente

Por ejemplo, si alimentamos el plano X aplicando una diferencia de potencial de 2.5 V entre los terminales X+ y X- y presionamos aproximadamente sobre la zona media entre los electrodos X, leeremos una diferencia de potencial entre los terminales Y+ y Y- de 1.25 V. Por lo tanto, esta tensión es proporcional a la tensión aplicada entre X+ y X- como resultado del divisor resistivo equivalente de la figura 14. De esta forma, podemos obtener la coordenada X del lugar en el que se ha pulsado. De igual modo, aplicando tensión entre los terminales Y+ y Y- podemos calcular la coordenada Y midiendo en el plano X. También es posible medir la presión con la que se ha tocado el panel, y normalmente se denotan estas medidas como “Z1” and “Z2”.

TSLIB – Librería para el panel táctil

TSLib es una librería que permite leer el punto que ha presionado el usuario en una pantalla táctil. Esta librería es gratuita, open source y además se utiliza en las librerías Qt que emplearemos más adelante para el desarrollo de aplicaciones gráficas con interfaz de usuario. Para lograr compilar e instalar esta librería para plataforma ARM seguiremos los siguientes pasos:

1. **Descargar TSLib**
   Podemos descargar los ficheros fuente de la página oficial de la librería:

http://tslib.berlios.de/
http://prdownload.berlios.de/tslib/tslib-1.0.tar.bz2

2. Descargar e instalar *autoconf*

El script de configuración de TSLib necesita una herramienta llamada *autoconf*, por lo que nos aseguraremos de que la tenemos instalada en nuestro ordenador de desarrollo antes de continuar. Si no la tuviéramos, se puede obtener desde el repositorio de forma estándar escribiendo:

```
> sudo apt-get install autoconf
```

Al terminar de instalar el paquete *autoconf*, el instalador nos sugiere que instalemos los siguientes paquetes adicionales:

- autoconf2.13
- autobook
- autoconf-archive
- gnu-standards
- autoconf-doc
- libtool
- gettext

Únicamente instalaremos el paquete *libtool*, que es el realmente imprescindible:

```
> sudo apt-get install libtool
```

3. Compilar TSLib

```
> cd tslib-1.0
> ./autogen.sh
> export ac_cv_func_malloc_0_nonnull=yes
> ./configure CC=arm-none-linux-gnueabi-gcc CXX=arm-none-linux-
gnueabi-g++ -host=arm-none-linux-gnueabi -target=arm-none-linux-
gnueabi -enable-static=no -enable-shared=yes
> make -k
> sudo make -k install
```

Para que TSLIB proporcione lecturas correctas para la pantalla LCD de 7'' suministrada con la DevKit8000 (originalmente se entregaba una pantalla de 4.3'' con el kit), es necesario realizar unos pequeños cambios en la especificación de la placa que viene en el código fuente del kernel. Para ello, hay que abrir el siguiente fichero:

$HOME/projects/linux-2.6.28-omap/arch/arm/mach-omap2/board-omap3devkit8000.c

Y dejar la configuración del controlador táctil de la siguiente forma:

```
struct ads7846_platform_data ads7846_conf = {
      .x_max                        = 0x0fff,
      .y_max                        = 0x0fff,
//    .x_plate_ohms                 = 180,
//    .pressure_max                 = 255,
      .debounce_max                 = 10,
      .debounce_tol                 = 5,
      .debounce_rep                 = 1,
      .get_pendown_state      = ads7846_get_pendown_state,
      .keep_vref_on           = 1,
      .settle_delay_usecs           = 150,
};
```

Guardamos y cerramos el fichero, recompilamos el kernel y lo grabamos en la memoria NAND.

PROCESO DE CALIBRACIÓN

Antes de poder usar la pantalla táctil y dar por buenas las lecturas, es necesario efectuar una calibración. Para ello disponemos de una herramienta que se compila junto con las librerías, denominada *ts_calibrate*. Ejecutaremos esta utilidad en la placa y veremos en pantalla una cruz sobre la que debemos de pinchar. Esto se repite 5 veces: una para cada esquina de la pantalla más una última lectura en el punto central de la pantalla. Tras esto, se generará el fichero */etc/ts.conf,* que guarda la información de calibración, y ya no será necesario volver a calibrar la pantalla para posteriores aplicaciones.

Figura 15: Utilidad de calibración ts_calibrate

VALIDACIÓN DE LA CALIBRACIÓN

Para comprobar que la calibración ha tenido éxito, se incluye otra herramienta denominada *ts_test*. Al ejecutar esta utilidad veremos como podemos dibujar sobre la pantalla una línea sencilla o bien pulsar sobre dos botones que ejemplo que hay dispuestos en la parte superior de la pantalla. Tras comprobar que la pantalla táctil funciona bien y el controlador reporta las coordenadas correctamente, procederemos a compilar las librerías gráficas Qt de Nokia, que se explica a continuación.

Figura 16: Utilidad de verificación ts_test

QT LIBRARIES FOR EMBEDDED LINUX – LICENCIA LGPL

Las librerías Qt de Nokia son una potente herramienta a la hora de escribir aplicaciones que necesitan interfaz gráfica de usuario. Son un conjunto de librerías que integran todo lo necesario, como por ejemplo botones, listas, etiquetas, gestor de ventanas y otros muchos controles gráficos.

Para este trabajo se ha descargado y compilado la versión de estas librerías para Embedded Linux. El proceso de compilación e instalación se explica a continuación:

1. Descargar las librerías para Embedded Linux

Podremos descargar el código fuente de los siguientes enlaces de Internet:

http://qt.nokia.com/downloads/embedded-linux-cpp
http://qt.nokia.com/downloads/downloads#lgpl

Concretamente para este trabajo, se ha descargado la version 4.5.1 que corresponde con el fichero: ***qt-embedded-linux-opensource-src-4.5.1.tar.gz***

2. QT: Compilar las librerías para ARM

Editar el fichero *qmake.conf* ubicado en el directorio *qt-embedded-linux-opensource-src-4.5.1/mkspecs/qws/linux-arm-g++/* para que queden las principales variables de la siguiente manera:

```
QMAKE_CC                = arm-none-linux-gnueabi-gcc
QMAKE_CXX           = arm-none-linux-gnueabi-g++
QMAKE_LINK          = arm-none-linux-gnueabi-g++
QMAKE_LINK_SHLIB    = arm-none-linux-gnueabi-g++

QMAKE_AR                = arm-none-linux-gnueabi-ar cqs
QMAKE_OBJCOPY       = arm-none-linux-gnueabi-objcopy
QMAKE_RANLIB            = arm-none-linux-gnueabi-ranlib
```

Tras guardar los cambios, ya podemos ejecutar el script de configuración:

```
> ./configure -embedded arm -no-armfpa -little-endian -qt-gfx-
transformed -qt-gfx-linuxfb -nomake demos -nomake examples  -no-svg
-no-phonon -no-qt3support -no-feature-CURSOR -qt-mouse-tslib -L
/usr/local/lib -I /usr/local/include -opensource
```

Cuando haya acabado de ejecutar dicho script, veremos en pantalla un mensaje similar al siguiente, indicando que ya está listo para compilar:

Qt is now configured for building. Just run 'make'.
Once everything is built, you must run 'make install'.
Qt will be installed into /usr/local/Trolltech/QtEmbedded-4.5.1-arm
To reconfigure, run 'make confclean' and 'configure'.

Por tanto, ya podemos compilar, ejecutando:

```
> make
```

El proceso de compilación es largo y en una máquina moderna suele llevar un mínimo de 50 minutos, aunque puede extenderse hasta algo más de 1 hora y 30 minutos. Durante este intervalo de tiempo nos aparecerán en pantalla muchos mensajes según vaya avanzando el proceso de compilación. Al finalizar, teclearemos:

```
> sudo make install
```

Tras esto, a pesar de algunos errores del final a los que no debemos prestar atención, ya tendremos las librerías compiladas para nuestra placa DevKit8000. Ahora copiaremos las principales librerías en el sistema de ficheros de la placa, junto con algunas fuentes, puesto que serán necesarios su presencia para poder ejecutar las aplicaciones que hagan uso de ellas. En concreto necesitaremos las siguientes del directorio */usr/local/Trolltech/QtEmbedded-4.5.1-arm/lib*, que deberemos renombrar a su versión mayor, además de los ficheros de fuentes tipográficas:

- libQtCore.so.4.5.1 → libQtCore.so.4
- libQtGui.so.4.5.1 → libQtGui.so.4
- libQtNetwork.so.4.5.1 → libQtNetwork.so.4
- /usr/local/Trolltech/QtEmbedded-4.5.1-arm/lib/fonts/*

En la DevKit8000, los anteriores ficheros deben de quedar en el mismo directorio que estaban originalmente instalados:

```
> mkdir -p /usr/local/Trolltech/QtEmbedded-4.5.1-arm/lib/fonts
> cp /projects/QT/*.so.4 /usr/local/Trolltech/QtEmbedded-4.5.1-
arm/lib
> cp /projects/QT/fonts/* /usr/local/Trolltech/QtEmbedded-4.5.1-
arm/lib/fonts
```

También se necesita añadir el directorio de la nueva librería al *PATH* del sistema:

```
> export PATH=/usr/local/Trolltech/QtEmbedded-4.5.1-arm/lib:$PATH
> export QWS_MOUSE_PROTO=Tslib:/dev/input/touchscreen0
> chmod a+rw /dev/input/event2
```

Para que cargue en memoria estas librerías Linux al arrancar el sistema, se deben de añadir al fichero de configuración ubicado en */etc/ld.so.conf* , donde añadiremos una línea como la siguiente:

```
/usr/local/Trolltech/QtEmbedded-4.5.1-arm/lib
```

Grabamos, cerramos y luego ejecutamos el siguiente comando para refrescar las librerías cargadas:

```
> ldconfig
```

Por último ya sólo nos hará falta la librería estándar de C++ (*libstdc++.so.6)* que la cogeremos ya compilada de las herramientas de CodeSourcery ubicada en el directorio:

```
$HOME/CodeSourcery/Sourcery_G++_Lite/arm-none-linux-
gnueabi/libc/usr/lib
```

, y la copiaremos al directorio */lib* de la placa.

INSTALACIÓN DEL ENTORNO DE DESARROLLO QTCreator

Nokia pone a nuestra disposición un completo entorno de desarrollo, denominado QtCreator, y que permite desarrollar aplicaciones gráficas empleando las librerías Qt de forma rápida y sencilla. Antes de poder instalar el IDE, es necesario instalar unos paquetes previos:

```
> sudo apt-get install libglib2.0-dev libSM-dev libxrender-dev
libfontconfig1-dev libxext-dev
```

Una vez tengamos todo lo anterior, procederemos a descargar de Internet de la página web de Nokia el instalador de QtCreator. Es necesario conceder permiso de ejecución bajo Ubuntu Linux previamente, y a continuación se puede instalar de forma normal:

```
> chmod +x qt-sdk-linux-x86-opensource-2009.02.bin
> ./qt-sdk-linux-x86-opensource-2009.02.bin
```

Para ver el proceso de instalación paso a paso, puede consultarse el anexo III.3 en donde se han incluido capturas de pantalla del proceso. Cuando haya finalizado el instalador, abriremos la aplicación Qt Creator y necesitaremos configurar el entorno para poder compilar para nuestra placa ARM:

- Iremos al menú Tools->Options, y se nos abrirá la siguiente ventana:

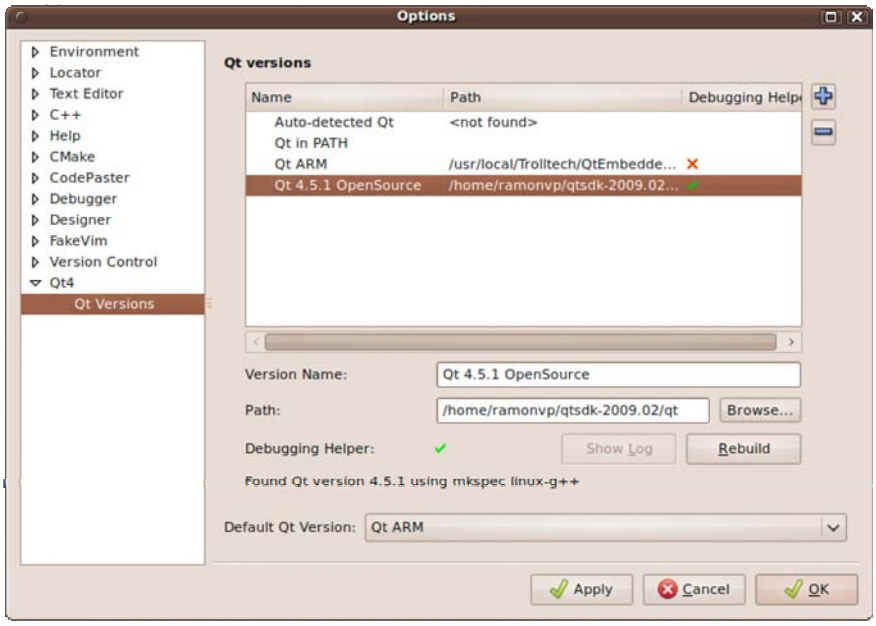


Figura 17: Configuración del compilador en Qt Creator

- En el panel izquierdo, seleccionaremos Qt4->Qt Versions.
- En la parte derecha aparecerá una lista, a la que añadiremos con el botón “+” ubicado arriba a la derecha nuestra propia configuración.
- Pulsamos “+” y aparecerá una nueva línea en la lista, en cuya columna Name podemos leer “<specify a name>” y en la columna Path “<specify a path>”.
- La seleccionamos con el ratón y en la parte inferior de la pantalla escribiremos en la casilla Version Name un nombre identificativo, como por ejemplo “Qt ARM”
- En la casilla Path escribiremos la ruta al directorio donde hemos dejado compiladas las librerías en el paso anterior, que seguramente sea similar a “/usr/local/Trolltech/QtEmbedded-4.5.1-arm”
- Inmediatamente debería de aparecer bajo el rótulo Debugging Helper el mensaje: “Found Qt version 4.5.1. using mkspec /usr/local/Trolltech/ QtEmbedded-4.5.1-arm/mkspecs/qws/linux-arm-g++”
- En el menú desplegable, seleccionaremos la opción recién creada, es decir “Qt ARM”
- Por último, aceptamos los cambios pinchando el botón “Ok”

Ya está todo preparado para poder realizar nuestra primera prueba con el entorno gráfico. Para ello iremos a la sección “Edit” en la parte izquierda de la pantalla, y haremos doble-click sobre el fichero llamado “mainwindow.ui”. Esto abrirá el editor de interfaz gráfica. Como prueba simple podemos arrastrar un botón normal (Push Button) de la paleta de controles disponibles. Tras esto, vamos al menú Build->Build All, nos pedirá que salvemos los cambios a fichero y finalmente compilará el ejecutable. El resultado de la compilación, así como cualquier mensaje de error que pueda surgir, lo podremos ver en la parte inferior de la pantalla, pinchando sobre el botón “Compile Output”.

Una vez tengamos nuestro ejecutable compilado y linkado, sólo bastará acceder por NFS desde la placa DevKit8000 y ejecutarlo con el parámetro –qws:

```
>./pruebaqt –qws
```

Se debería de poder ver el botón en la pantalla LCD de la placa. Para finalizar el programa, al no haber previsto ningún código de salida, deberemos pulsar la combinación CTRL+C para forzar el fin del proceso.

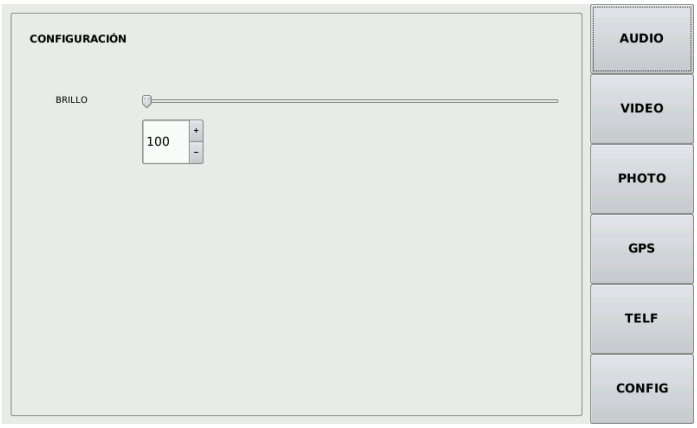


Figura 18: Ejemplo de interfaz gráfica con Qt

2.4.3. Audio en la placa DevKit8000

El subsistema de audio es precisamente uno de los sistemas que no viene integrado en el propio OMAP3530, pero se ha decidido comentarlo en esta sección dada la importancia que tiene el audio en un sistema multimedia.

El responsable del audio realmente es un chip que acompaña en la placa denominado TPS65930, y que también es del fabricante Texas Instruments. De hecho, este circuito integrado fue diseñado específicamente como chip de acompañamiento al OMAP3530. El TPS65930 cuentas con las siguientes características:

- Alimentación:
  - 3 conversores DC-DC step-down muy eficientes
  - 4 reguladores de tipo low-dropout para clocks y periféricos
  - Sistema “SmartReflex” para la gestión dinámica de la tensión

- Audio
  - Entrada diferencial principal para micrófono
  - Entrada auxiliar mono/entrada FM
  - External predrivers for class D (stereo)
  - Interfaz de datos para audio TDM /I2S
  - Automatic Level Control (ALC)
  - Mezclado digital y analógico
  - Conversor DAC de 16-bit stereo (96, 48, 44.1, and 32 kHz y derivadas)
  - Conversor ADC de 16-bit stereo (48, 44.1, and 32 kHz y derivadas)

- USB
  - USB 2.0 On-The-Go (OTG)
  - Proporciona alimentación al bus USB (5-V)
  - Cumple con CEA-2011 (Consumer Electronics Association)
  - Cumple con CEA-936A: Mini-USB

- Características adicionales
  - Circuito driver para dos LEDs externos
  - Conversor MADC de 10 bits de 2 entradas externas
  - Reloj para tiempo real (RTC)
  - Control a través del bus I2C
  - Control térmico y detección de encapsulado caliente
  - Soporte para un teclado matricial hasta 6 × 6
  - Control para vibrador externo
  - 15 pines de entrada-salida de propósito general (GPIOs)

En el anexo II podemos ver cómo se interconecta el TPS65930 al OMAP35350. En concreto se observa que la información de audio digital se transmite a

través de la interfaz I2S del TPS65930, que corresponde con el puerto McBSP2. En cambio la información de control, como por ejemplo frecuencia de muestreo, stereo o mono, etc., se transmite a través de otra interfaz serie diferente, en este caso de tipo I2C.

AUDIO EN LINUX

Hasta los kernels 2.4.x, se empleaba una librería de audio denominada OSS (Open Sound System) como modelo para crear los drivers. Sin embargo, a partir de los kernels 2.6.x, se ha adoptado otro modelo para el audio en Linux denominado ALSA (Advanced Linux Sound Architecture). En la placa DevKit8000, tenemos compilado por defecto en el kernel el driver para la plataforma ALSA.

El driver del códec de audio TWL4030 se encuentra en:

$HOME/projects/linux-2.6.28-omap/sound/soc/codecs/twl4030.c

El TWL4030 era un módulo que ofrecía unas características muy parecidas al TPS65930, pero que ahora ya ha quedado obsoleto y ya no se fabrica. Sin embargo, al ser compatibles pin a pin, el controlador que utiliza Linux para comunicarse con este chip es el mismo que para el TWL4030.

COMANDOS ÚTILES

Para listar los dispositivos para reproducción de sonido:

```
aplay -l
**** List of PLAYBACK Hardware Devices ****
card 0: omap3devkit9000 [omap3devkit9000], device 0: TWL4030 twl4030-
I2S-0 []
  Subdevices: 1/1
  Subdevice #0: subdevice #0
```

Para listar los dispositivos para grabación de sonido:

```
arecord -l
**** List of CAPTURE Hardware Devices ****
card 0: omap3devkit9000 [omap3devkit9000], device 0: TWL4030 twl4030-
I2S-0 []
  Subdevices: 1/1
  Subdevice #0: subdevice #0
```

LIBRERIAS PARA AUDIO

Como se ha comentado anteriormente, la librería que se emplea actualmente para el manejo de audio en Linux se denomina ALSA y podemos descargar el código fuente de esta librería directamente desde la página web oficial de ALSA. No confundir la librería de usuario con la librería de desarrollador, orientada ésta última a la creación de drivers para dispositivos de audio para Linux.

- **ALSA-LIB**

http://www.alsa-project.org/main/index.php/Download

ftp://ftp.alsa-project.org/pub/lib/alsa-lib-1.0.22.tar.bz2
(790 KB)

También conviene descargar un paquete de utilidades que también compilaremos para nuestra placa. Son las denominadas ALSA-UTILS y aunque algunas ya vienen por defecto con la placa DevKit8000, conviene disponer del código fuente puesto que sirven como un excelente ejemplo como referencia para dominar los aspectos básicos de la librería ALSA.

- **ALSA-UTILS (aplay, arecord)**

ftp://ftp.alsa-project.org/pub/utils/alsa-utils-1.0.22.tar.bz2
(1.0 MB)

```
> ./configure --host=arm-none-linux-gnueabi --prefix=$TOOLCHAIN --
disable-alsamixer --disable-xmlto
> make
> make install
```

USO DE LAS FUNCIONES ALSA-LIB

La librería ALSA ofrece múltiples funciones para controlar cada uno de los parámetros relativos al dispositivo hardware de audio que tengamos instalado en nuestro sistema. Concretamente, existen los siguientes grupos de funciones:

- **Control interface**: sirve para manipular registros de propósito general de la tarjeta de sonido y sondear el hardware disponible

- **PCM interface**: este es el interfaz principal para manejar la reproducción y captura de audio. Será el que más habitualmente usaremos en aplicaciones que empleen audio

- **Raw MIDI interface**: interfaz de soporte a MIDI (Musical Instrument Digital Interface) para controlar teclados electrónicos y otros instrumentos de música con este conector. El programador es responsable de gestionar los sincronismos y los protocolos empleados

- **Timer interface**: proporciona acceso a temporizadores del dispositivo de audio para poder sincronizar eventos de sonido

- **Sequencer interface**: un interfaz para MIDI de un nivel superior que el RAW, que maneja automáticamente el protocolo y sincronismo
- **Mixer interface**: permite el acceso al dispositivo mezclador hardware que encamina las señales de audio y los niveles de control de volumen. El mezclador utiliza la interfaz ofrecida por el módulo de Control

FLUJO HABITUAL PARA REPRODUCIR AUDIO

1. Abrir el dispositivo de reproducción de audio:

```
int  snd_pcm_open(snd_pcm_t  **pcm,
                  const char *name,
                  snd_pcm_stream_t  stream,
                  int mode)
```

Parámetros:

- *snd_pcm_t **pcm*: manejador que usaremos en todas las llamadas para hacer referencia al dispositivo que abrimos
- *const char *name*: nombre del dispositivo que queremos abrir. Lo obtenemos consultando el listado que se obtiene con el comando aplay –L, aunque por simplicidad se puede usar “default”
- *snd_pcm_stream_t stream*: SND_PCM_STREAM_PLAYBACK para indicar que abrimos el dispositivo como salida
- *int mode*: modo de apertura, usaremos 0 para indicar que no debe bloquear

2. Obtener los parámetros de configuración de hardware

```
int snd_pcm_hw_params_any (   snd_pcm_t  * pcm,
                              snd_pcm_hw_params_t *  params)
```

Parámetros:

- *snd_pcm_t **pcm*: manejador de dispositivo
- *snd_pcm_hw_params_t * params*: puntero a una estructura de datos que almacenará la configuración de los parámetros hardware. Esta estructura se debe obtener con una llamada a snd_pcm_hw_params_alloca(), que nos reservará memoria e inicializará la estructura

3. Indicar orden en el que se entregan los datos de audio

```
int snd_pcm_hw_params_set_access(  snd_pcm_t *pcm,
                                   snd_pcm_hw_params_t *params,
                                   snd_pcm_access_t _access);
```

Parámetros:

- *snd_pcm_t **pcm*: manejador de dispositivo
- *snd_pcm_hw_params_t *params*: puntero a una estructura de parámetros hardware que queremos manipular
- *snd_pcm_access_t _access*: orden deseado. En la mayoría de aplicaciones se usa SND_PCM_ACCESS_RW_INTERLEAVED para indicar que las muestras del canal izquierdo y derecho van alternadas en el tiempo

4. Indicar formato de las muestras de audio

```
int snd_pcm_hw_params_set_format(  snd_pcm_t *pcm,
                                   snd_pcm_hw_params_t *params,
                                   snd_pcm_format_t val);
```

Parámetros:

- *snd_pcm_t **pcm*: manejador de dispositivo
- *snd_pcm_hw_params_t *params*: puntero a una estructura de parámetros hardware que queremos manipular
- *snd_pcm_format_t val*: formato en el que enviamos las muestras, que en este trabajo se ha empleado un formato de 16bits por muestras con signo. A su vez, el descodificador nos deja en memoria estas muestras en formato Little Endian, es decir, el byte de menor peso en primer lugar. Para esto hay declarado un tipo enum que corresponde con nuestra configuración y es: SND_PCM_FORMAT_S16_LE

5. Indicar frecuencia de muestreo

```
int snd_pcm_hw_params_set_rate_near (
                    snd_pcm_t *pcm,
                    snd_pcm_hw_params_t *params,
                    unsigned int *val,
                    int *dir);
```

Esta función va a intentar establecer la frecuencia de muestreo al valor más próximo posible al solicitado. Algunas veces el real será el solicitado, y otras veces no será posible. Para las frecuencias de muestreo estándar (22050, 44100, 48000) no habrá ningún problema. Si fuera una frecuencia intermedia de estos valores, entonces con el parámetro *dir* especificaremos si queremos que la real se ajuste por encima (+1), por debajo (-1) o bien intente ser exacta (0).

Parámetros:

- *snd_pcm_t **pcm*: manejador de dispositivo
- *snd_pcm_hw_params_t *params*: puntero a una estructura de parámetros hardware que queremos manipular
- unsigned int val: frecuencia de muestra deseada en Hz. Para nuestro caso, la mayoría de pruebas realizadas se hicieron con 44100 Hz
- *int *dir*: 0 para indicar que queremos exactamente la frecuencia de muestreo especificada en el parámetro *val*

6. Indicar número de canales de audio

```
int snd_pcm_hw_params_set_channels(snd_pcm_t *pcm,
                                   snd_pcm_hw_params_t *params,
                                   unsigned int val);
```

Parámetros:

- *snd_pcm_t **pcm*: manejador de dispositivo
- *snd_pcm_hw_params_t *params*: puntero a una estructura de parámetros hardware que queremos manipular
- unsigned int val: número de canales de audio, en nuestro caso emplearemos 2 canales, el izquierdo y derecho

7. Pasar esta información de configuración de hardware al driver y liberar

```
int snd_pcm_hw_params( snd_pcm_t *pcm,
                       snd_pcm_hw_params_t *params);
```

Parámetros:

- *snd_pcm_t **pcm*: manejador de dispositivo
- *snd_pcm_hw_params_t *params*: puntero a la estructura de parámetros hardware que queremos volcar sobre el driver

Dado que no necesitamos configurar ningún parámetro de hardware más, ya podemos liberar la memoria ocupada por esta estructura, para lo que llamaremos a la función:

void snd_pcm_hw_params_free(snd_pcm_hw_params_t *obj);

Parámetros:

- *snd_pcm_hw_params_t *obj*: puntero a la estructura de datos que queremos liberar en memoria

8. Preparar el dispositivo para iniciar la reproducción

int snd_pcm_prepare(snd_pcm_t *pcm);

Parámetros:

- *snd_pcm_t **pcm*: manejador de dispositivo que debe de activarse y prepararse para ser usado

9. Mandar las muestras de audio al dispositivo

La siguiente función es la encargada de ir enviando las muestras que tenemos en memoria al driver para que éste pueda a su vez enviárselas al conversor DAC para su reproducción.

```
snd_pcm_sframes_t snd_pcm_writei(   snd_pcm_t *pcm,
                                    const void *buffer,
                                    snd_pcm_uframes_t size);
```

Parámetros:

- *snd_pcm_t **pcm*: manejador de dispositivo
- *snd_pcm_hw_params_t *params*: puntero a una estructura de parámetros hardware que queremos manipular
- unsigned int val: número de canales de audio, en nuestro caso emplearemos 2 canales, el izquierdo y derecho

10. Al finalizar, cerrar el dispositivo

int snd_pcm_close (snd_pcm_t *pcm);

Parámetros:

- *snd_pcm_t **pcm*: manejador del dispositivo que queremos cerrar

Importante:
Como detalle, recordar que para compilar aplicaciones contra la librería alsa-lib hay que añadir *–lasound* en el comando gcc correspondiente e incluir la referencia a la cabecera base en el código fuente de nuestra aplicación:

```
#include <alsa/asoundlib.h>
```

2.5. COMUNICACIÓN CON EL DSP INTEGRADO DEL OMAP3530

El procesador OMAP3530 incorpora un DSP TMS320C64x+, también de Texas Instruments. Dicho DSP va destinado a tareas de cálculos más complejos, como tratamiento de la señal, codificación y descodificación, filtrado de señales y otras operaciones matemáticas, para las cuales está diseñado muy eficientemente.

Dicho DSP actúa de esclavo respecto al microprocesador principal, en el sentido de que recibe los datos y la orden de las operaciones a realizar y al terminar devuelve el resultado, quedando a la espera de recibir nuevas órdenes. El mecanismo de comunicación entre ambos procesadores es a través de lo que se denominan “mailboxes”, que son un tipo de registros internos de la CPU dedicados para este fin.

El manejo de estos registros es complicado y tedioso y es por ello que para poder comunicarnos con el DSP desde el GPP (General Purpose Processor), Texas Instruments nos ofrece una librería que nos va a ahorrar mucho trabajo. La arquitectura del sistema que propone DSPLink es la siguiente:

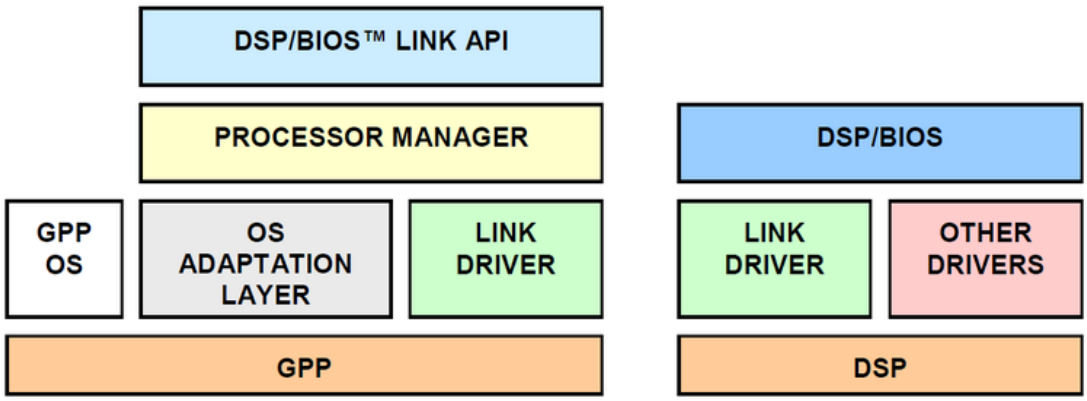


Figura 19: Arquitectura software de DSP/BIOS LINK

2.5.1. Modelo de funcionamiento

Para empezar a escribir programas que utilicen DSP/BIOS LINK, es importante conocer y seleccionar los módulos DSPLINK más apropiados según las necesidades de nuestra aplicación. Según los requisitos de la aplicación, podremos usar todos o únicamente un subconjunto de las características que ofrece DSPLINK. Los módulos que ofrece el sistema DSPLINK son los siguientes:

PROC

Este componente es siempre necesario para todas las aplicaciones que usen DSP/BIOS LINK. Proporciona la funcionalidad básica para establecer la configuración, gestionar el arranque, el control y la comunicación con los otros procesadores del sistema.

NOTIFY

Este componente pueda usarse para mensajería y transferencia de datos si:

1. La información a traspasar entre procesadores es de longitud 32 bits
2. Se necesita prioridad en las notificaciones. Por ejemplo, un evento con baja prioridad (p.ej. 30) puede usarse para enviar punteros a búfers, mientras que otro evento con prioridad más alta (p.ej. 5) puede emplearse para el envío de comandos.
3. Las notificaciones serán infrecuentes. Si se envían múltiples notificaciones muy rápidamente para el mismo evento, cada intento permanece a la espera hasta que el evento previo ha sido atendido. Esto puede provocar un funcionamiento poco eficiente.
4. Múltiples clientes necesitan registrarse para recibir el mismo evento de notificación. Cuando se recibe la notificación de evento, la misma notificación con una pequeña sobrecarga de datos (*payload*) se reenvía a todos los clientes registrados para esa misma notificación.

MSGQ

Este componente puede emplearse para transferencia de mensajes y de datos si se cumplen las siguientes condiciones:

1. La aplicación necesita un solo lector y múltiples escritores
2. Se necesitan datos de longitudes superiores a 32 bits, que se enviarán en estructuras de mensaje definidas por el usuario
3. Se necesitan mensajes de tamaño variable
4. Lector y escritor manejan los mismos tamaños de buffer
5. Se necesita intercambiar mensajes con frecuencia. En este caso los mensajes esperan en una cola y no es necesario esperar a que el evento anterior finalice.
6. Se desea poder esperar cuando la cola está vacía. Este comportamiento es el natural del protocolo MSGQ y no es necesario añadir código desde la aplicación. Si se llama a la función *MSGQ_get()* cuando la cola está vacía, se espera hasta recibir un mensaje. Si se usara el módulo NOTIFY, la aplicación debería registrar una función callback o bien esperar en un semáforo a que le señalicen.
7. Se desea poder mover la cola de mensajes entre procesadores. En este caso la función *MSGQ_locate()* se hace cargo internamente de reubicar la cola y el código de la aplicación que envía mensajes a esta cola no necesita modificarse.
8. Se desea tener comunicación de un DSP a otro DSP. En este caso, el módulo MSGQ permite la ejecución de varios módulos independientes para poder comunicar entre los DSP.

MPLIST

Este componente puede emplearse para transferencias de mensajes y de datos si:

1. La aplicación necesita múltiples lectores y escritores.
2. La aplicación desea procesar los paquetes recibidos en otro orden del de llegada. Esto es no es posible con MSGQ ni con NOTIFY. Con NOTIFY, el lector puede alterar la lista compartida y escoger cualquier buffer.
3. Se desea poder marcar un buffer específico como de alta prioridad. El emisor puede colocar el buffer o mensaje transmitido al principio de la cola, en vez de al final que sería lo habitual. Se facilitan unas APIs que permiten alterar el orden de la lista e insertar un elemento en la posición deseada.
4. Se desea flexibilidad a la hora de notificar los eventos de enviado y recibido. La aplicación puede usar el módulo NOTIFY para enviar y recibir notificaciones según sus necesidades. Esto puede resultar en un mejor rendimiento de la aplicación al reducir el número de interrupciones que se generan. A cambio, es necesario agregar más líneas de código para hacer algo que ya es inherente en MSGQ.
5. Lector y escritor operan sobre los mismos tamaños de buffer.
6. Los datos a intercambiar son de longitud superior a 32 bits que se enviarán usando estructuras de mensajes definidas por la aplicación.
7. Se requieren búfers de tamaño variable destinados a datos y mensajes.
8. Se necesita enviar mensajes e información frecuentemente. En este caso, los mensajes se encolan directamente.

CHNL

Este módulo puede emplearse para transferir data si:

1. Se necesita un único lector y un único escritor.
2. Los búfers de datos son de tamaño fijo.
3. Lector y escritor operan en el mismo tamaño de buffer
4. Es necesario utilizar drivers SIO para otros periféricos al mismo tiempo que la comunicación con el DSP.
5. Se requiere un protocolo sencillo para streaming de datos. Para estos casos el modulo CHNL proporciona un protocolo sencillo de demanda-suministro de datos. La aplicación sólo necesita proporcionar búfers vacíos o llenos a ambos procesadores, y éstos se van intercambiando cuando quedan disponibles entre procesadores del mismo canal. Si un buffer no está disponible, se dispone de mecanismos de espera y notificación cuando se reclama un búfer.
6. Múltiples búfers pueden ser fácilmente encolados para un mayor rendimiento.

RingIO

Este componente puede emplearse para transferir mensajes y de datos si:

1. Sólo se necesita un único lector y escritor
2. Debe ser posible enviar y recibir datos, al igual que atributos/mensajes asociados a los datos
3. Escritor y lector necesitan poder seguir ejecutando con independencia del otro. El tamaño del buffer empleado por el escritor puede ser diferente del tamaño que usa el escritor
4. Lector y escritor pueden liberar menos memoria de la que reservaron inicialmente o pueden solicitar más memoria sin liberar ninguna
5. Las aplicaciones tienen necesidades de notificación distintas. La aplicación puede minimizar las interrupciones escogiendo el tipo de notificación más apropiado.
6. Se desea tener la posibilidad de eliminar datos no utilizados del buffer
7. Se desea poder vaciar el contenido de un buffer entero. Esto puede ser interesante en el caso de una reproducción de un fichero multimedia y se interrumpe para empezar a descodificar un fichero nuevo.

A continuación se describen 3 diferentes escenarios con necesidades de comunicación con el DSP muy diferentes. Para cada escenario se sugiere una solución de diseño así como el conjunto de módulos DSPLINK a usar.

1. Búfer estático con comunicación de control mínima con el DSP

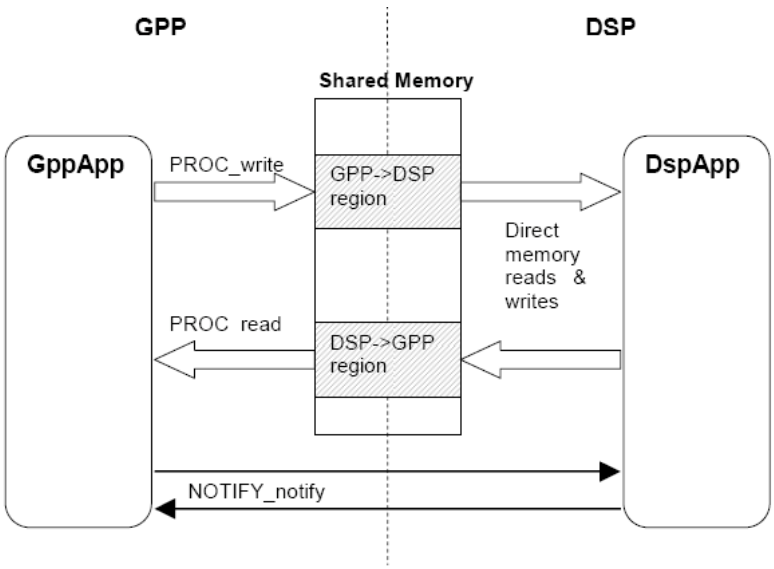


Figura 20: Escenario con búfer estático entre GPP-DSP

- Se emplea el modulo PROC para arrancar el DSP. El fichero ejecutable del DSP reside en el sistema de ficheros del GPP.
- En tiempo de compilación se pueden reservar dos regiones de memoria a través del fichero dinámico de configuración (CFG_<PLATFORM>.c). En el lado del DSP, hay que realizar una configuración similar en el

fichero TCF para reservar la memoria. Una región de memoria puede emplearse para las transferencias en sentido GPP->DSP, y la otra región para el sentido DSP->GPP. Dado que la memoria está reservada estáticamente, tanto el GPP como el DSP saben las direcciones de inicio y los tamaños de ambas memorias.

- El modulo NOTIFY se puede usar para enviar mensajes de control de 32 bits entre GPP y DSP.

2. Búfer dinámico con comunicación de control mínima con el DSP

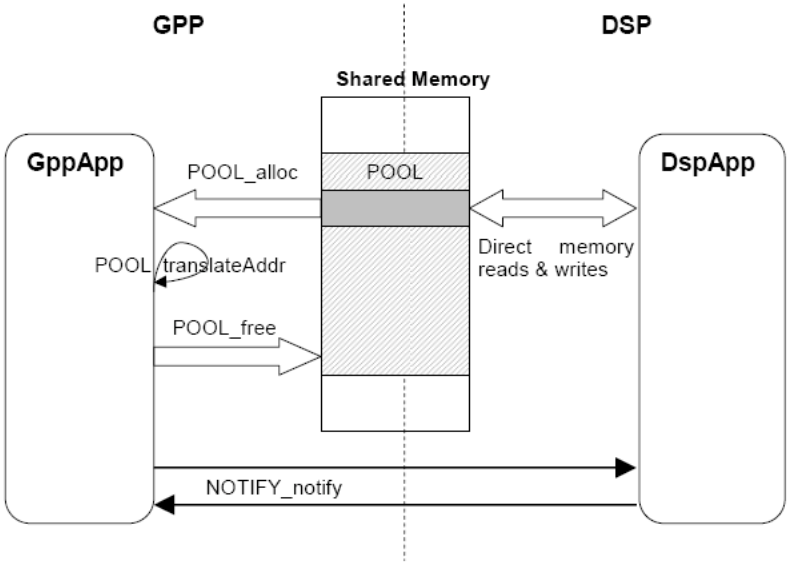


Figura 21: Escenario con búfer dinámico entre GPP-DSP

- Se emplea el módulo PROC para arrancar el DSP. El fichero ejecutable del DSP reside en el sistema de ficheros del GPP.
- Se abre un espacio denominado POOL con la configuración del tamaño de los búfers que compartirán GPP y DSP
- Se reserva el espacio para los búfers del POOL según necesiten GPP Y DSP durante la fase de inicio.
- Si se ha reservado el búfer desde el GPP, la dirección de este búfer creado por la función *POOL_alloc*() puede traducirse al espacio de direcciones del DSP usando la función *POOL_translateAddr*().
- Se puede emplear el módulo NOTIFY para enviar las direcciones dde 32 bits de los buffers o intercambiar información de control entre GPP y DSP.
- Si el búfer se reserva desde el lado DSP, la dirección recibida puede traducirse en el lado GPP con la función *POOL_translateAddr*().
- Los búfers ya pueden ser usados por GPP y DSP para intercambiar información y datos. Se puede emplear el módulo NOTIFY para informar a los procesadores de que hay datos disponibles en el búfer o que el búfer ha sido vaciado y liberado.
- Si tanto GPP como DSP pueden acceder simultáneamente a los búfers y se necesita proporcionar mecanismos de exclusión mutua, podremos usar opcionalmente el módulo MPCS para proteger estos accesos.

3. Múltiples Búfers compartidos entre GPP y DSP

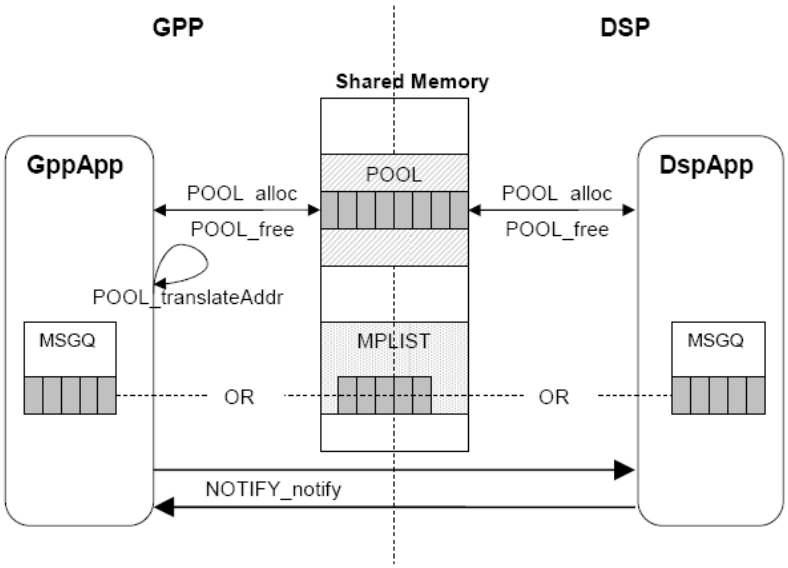


Figura 22: Escenario con múltiples búfers entre GPP-DSP

- Se emplea el módulo PROC para arrancar el DSP. El fichero ejecutable del DSP reside en el sistema de ficheros del GPP.
- Se abre un espacio denominado POOL con la configuración del tamaño de los búfers que compartirán GPP y DSP
- Se reserva el espacio para los búfers del POOL según necesiten GPP Y DSP durante la fase de inicio.
- Si se ha reservado el búfer desde el GPP, la dirección de este búfer creado por la función *POOL_alloc*() puede traducirse al espacio de direcciones del DSP usando la función *POOL_translateAddr*().
- Si los búfers tienen que enviarse con poca frecuencia, se puede usar el módulo NOTIFY para transmitir las direcciones de 32 bits (o cualquier otra información de control de 32 bits). Si uno de los procesadores necesita enviar múltiples búfers de una sola vez, se pueden usar los módulos MSGQ o MPLIST. Si se requiere enviar información adicional asociada al búfer, por ejemplo atributos del búfer, se puede definer una estructura de mensaje con estos atributos y usar el componente MSGQ o MPLIST para transmitir este mensaje al otro procesador.
- Si el búfer se reserva desde el lado DSP, la dirección recibida puede traducirse en el lado GPP con la función *POOL_translateAddr*().

2.5.2. Compilación de las librerías y módulos DSPLink

Primero descargaremos la última versión del DSP/BIOS (Built-In Operating System). Esto es un sistema operativo propio que se está ejecutando en el DSP, por lo que además de la fuente, también necesitaremos descargar las herramientas adecuadas para compilarlas.

- Descarga de la librería/driver DSPLINK:
http://software-dl.ti.com/dsps/dsps_public_sw/sdo_sb/targetcontent/DSPLink/
Descargaremos la versión DSPLink 1.64 for Linux, resultando el fichero:
dsplink_linux_1_64.tar.gz

- Descarga del sistema operativo del DSP/BIOS
http://focus.ti.com/docs/toolsw/folders/print/dspbios5.html
Aqui elegimos: DSP/BIOS 5.41.03.17
En la siguiente página elegimos descargar: 5.41.03.17 Linux
Y finalmente obtenemos el fichero: bios_setuplinux_5_41_03_17.bin (45 MB)

- Herramientas para compilar el DSPLink yBIOS
http://software-dl.ti.com/dsps/dsps_public_sw/sdo_sb/targetcontent/omap_l138/1_00/latest/index_FDS.html
Descargaremos el fichero: xdctools_setuplinux_3_16_01_27.bin (171 MB)

https://www-a.ti.com/downloads/sds_support/TICodegenerationTools/download.htm
Descargamos este fichero: ti_cgt_c6000_6.1.13_setup_linux_x86.bin (73.8 MB)

LOCAL POWER MANAGEMENT
http://software-dl.ti.com/dsps/dsps_registered_sw/sdo_sb/targetcontent/linuxutils/linuxutils_2_23/index.html
Fichero a descargar: local_power_manager_1_23_01.tar.gz (1.3 MB)

CODEC ENGINE
http://software-dl.ti.com/dsps/dsps_registered_sw/sdo_sb/targetcontent/CE/ce_2_24/index.html

Fichero a descargar: codec_engine_2_24_01.tar.gz

Nota: Es probable que sea necesario registrarse en la página web de Texas Instruments para poder acceder a las descargas.

Para los 3 ficheros anteriores, les damos permisos de ejecución:

```
> chmod +x bios_setuplinux_5_41_03_17.bin
> chmod +x xdctools_setuplinux_3_16_01_27.bin
> chmod +x ti_cgt_c6000_6.1.13_setup_linux_x86.bin
```

A continuación instalamos todo lo anterior según se muestra en el anexo III.4.

Al finalizar el proceso, hay que añadir a .bashrc las siguientes variables de entorno:

```
export
C6X_C_DIR=$HOME/ti/TI_CGT_C6000_6.1.13/include:$HOME/ti/TI_CGT_C6000
_6.1.13/lib
export DSPLINK=$HOME/projects/dsplink_linux_1_64/dsplink
```

Luego hay que editar los siguientes 3 makefiles:

$DSPLINK/make/Linux/omap3530_2.6.mk
$DSPLINK/gpp/src/Rules.mk
$DSPLINK/make/DspBios/c64xxp_5.xx_linux.mk

En los 3, habrá que poner los directorios de nuestro caso en las principales variables que veamos.

- Hay que añadir al path el directorio de los scripts, pero solo temporalmente para la configuración:

```
> export PATH =
$HOME/projects/dsplink_linux_1_64/dsplink/etc/host/scripts/Linux:$
PATH
```

A continuación hay que configurar los parámetros de la compilación, para lo cual:

```
> cd $HOME/projects/dsplink_linux_1_64/dsplink/config/bin
> perl dsplinkcfg.pl --platform=OMAP3530 --nodsp=1 --
dspcfg_0=OMAP3530SHMEM --dspos_0=DSPBIOS5XX --gppos=OMAPLSP --
comps=ponslrmc

> cd $DSPLINK/dsp
> $HOME/ti/xdctools/xdc clean
> $HOME/ti/xdctools/xdc .interfaces

> cd $DSPLINK/gpp
> $HOME/ti/xdctools/xdc clean
> $HOME/ti/xdctools/xdc .interfaces
```

Ya estamos listos para compilar. Pero antes habrá que modificar un fichero .h del código fuente de Linux, porque falta definir un pequeño detalle:

- Abrir $HOME/projects/linux-2.6.28-omap/arch/arm/include/asm/sizes.h
- Añadir el siguiente define en la sección que le pertoca:

  #define SZ_2K 0x00000800
- Guardar y salir

Ya podemos compilar todo lo del procesador principal:

```
> cd $DSPLINK/gpp/src/api
> make -s clean
> make -s debug
> make -s release
> cd $DSPLINK/gpp/src
> make -s clean
> make -s debug
> make -s release
> cd $DSPLINK/gpp/src/examples
> make -s clean
> make -s debug
> make -s release
```

Y a continuación compilamos el lado del DSP:

```
cd $DSPLINK/dsp/src
make -s release
make -s debug
cd $DSPLINK/dsp/src/samples
make -s release
make -s debug
```

Una vez compilado, movemos los ficheros necesarios al directorio compartido por NFS con la placa DevKit8000:

```
> mkdir /home/ramonvp/projects/dsplink
> cp export/BIN/Linux/OMAP3530/DEBUG/dsplinkk.ko
/home/ramonvp/projects/dsplink
> cp $DSPLINK/gpp/export/BIN/Linux/OMAP3530/DEBUG/messagegpp
/home/ramonvp/projects/dsplink
> cp
$DSPLINK/dsp/export/BIN/DspBios/OMAP3530/OMAP3530_0/DEBUG/message.ou
t /home/ramonvp/projects/dsplink
```

COMPILACION DE LOCAL POWER MANAGEMENT (LPM)

En primer lugar, descomprimimos el fichero que hemos descargado anteriormente de Internet, y lo pondremos todo en algún directorio que nos convenga:

```
cd $HOME/work
tar xvf ../Downloads/local_power_manager_1_23_01.tar.gz
cd local_power_manager_1_23_01/omap3530/lpm
```

Modificamos las siguientes 3 líneas del fichero Makefile de este directorio:

```
LINUXKERNEL_INSTALL_DIR = ${HOME}/work/linux-2.6.28-omap
MVTOOL_PREFIX = ${HOME}/CodeSourcery/Sourcery_G++_Lite/bin/arm-none-
linux-gnueabi-
DSPLINK_REPO = ${HOME}/work/dsplink_linux_1_64
```

Luego compilamos y ya dispondremos del módulo LPM para insertar en el kernel. Lo copiaremos como antes al directorio NFS:

```
> make
> cp lpm_omap3530.ko $HOME/projects/dsplink
```

Desde la placa:

```
> cp
/projects/local_power_manager_1_23_01/omap3530/lpm/lpm_omap3530.ko
./
> cp
/projects/local_power_manager_1_23_01/packages/ti/bios/power/test/bi
n/ti_platforms_evm3530/linux/debug/lpmON.x470uC ./DEBUG
> cp
/projects/local_power_manager_1_23_01/packages/ti/bios/power/test/bi
n/ti_platforms_evm3530/linux/debug/lpmOFF.x470uC ./DEBUG
> cp
/projects/local_power_manager_1_23_01/packages/ti/bios/power/test/bi
n/ti_platforms_evm3530/linux/release/lpmOFF.x470uC ./RELEASE
> cp
/projects/local_power_manager_1_23_01/packages/ti/bios/power/test/bi
n/ti_platforms_evm3530/linux/release/lpmON.x470uC ./RELEASE
>cp
/projects/local_power_manager_1_23_01/packages/ti/bios/power/test/bi
n/ti_platforms_evm3530/linux/debug/lpmON.x470uC /usr/bin
> cp
/projects/local_power_manager_1_23_01/packages/ti/bios/power/test/bi
n/ti_platforms_evm3530/linux/debug/lpmOFF.x470uC /usr/bin
```

CONFIGURACIÓN DE LA MEMORIA COMPARTIDA

Para que tanto la CPU como el DSP puedan acceder a las mismas regiones de memoria, se debe de configurar el tamaño de esta zona de memoria compartida. Este paso debe hacerse desde el gestor de arranque. Por lo tanto, en el U-Boot, hay que añadir un detalle en una variable de entorno, que corresponde con la variable que almacena los parámetros que le pasará al kernel cuando lo arranque:

➔ Añadir “mem=80M” a la variable de entorno <bootargs>

LIBRERÍA DSPLIB CON FUNCIONES YA IMPLEMENTADAS

DSPLIB es una librería gratuita de funciones matemáticas y de procesado desarrollada por Texas Instruments y que está optimizada para su ejecución sobre DSP. Soporta una amplia gama de procesadores DSP del propio fabricante, entre los que se encuentra el nuestro, que es el C64x+. DSPLib incorpora funciones tales como filtrados, convoluciones y FFT entre otras. Para una lista completa de las funciones incorporadas, puede consultarse el Anexo V.

Podemos descargar esta librería desde el siguiente enlace web:

http://focus.ti.com/docs/toolsw/folders/print/sprc265.html#Order%20Options
http://focus.ti.com/lit/sw/sprc834/sprc834.gz

Buscar el enlace rotulado: **C64x+DSPLIB Linux Install**

2.5.3. Fases de un programa con DSP

En general, los programas que necesitan usar el DSP como ayuda para realizar procesamiento de información, se suelen dividir en tres etapas:

Etapa 1: Fase de Inicialización
En esta primera fase, se crean todos los enlaces y se reservan los espacios en memoria necesarios para poder alojar los búfers de intercambio de información.

Etapa 2: Fase de Ejecución
Durante esta fase, ambos procesadores se estan intercambiando información, mensajes de control especificando qué hacer con los datos, y devolviendo los resultados.

Etapa 3: Fase de Finalización
En esta última etapa, se destruyen todos los caminos creados durante la fase de inicialización y se libera toda memoria que se hubiera reservado para el funcionamiento de la aplicación.

A continuación se comentará el ejemplo LOOP, añadiendo en el lado DSP el cálculo de una transformada FFT del búfer de recepción aprovechando la librería DSPLIB, cuyo listado completo de funciones ofrecidas se puede consultar en el Anexo IV. Esta aplicación realiza lo siguiente:

- GPP envía un búfer de datos al DSP
- GPP indica al DSP que los datos están disponibles
- DSP recoge los datos y realiza una FFT sobre éstos
- DSP devuelve los datos al GPP escribiendo a otro búfer
- GPP recoge el resultado de la FFT y finaliza

EN EL LADO GPP:

Fase de Inicialización

1. El cliente configura las estructuras de datos necesarias para acceder al DSP. A continuación se vincula al DSP identificado por DSP PROCESSOR IDENTIFIER.
2. Abre la pool para poder asignar espacio a los búfers de datos.
3. Carga el fichero ejecutable del DSP (loop.out) en el DSP.
4. Crea los canales CHNL_ID_INPUT y CHNL_ID_OUTPUT por donde se transmitirán los datos.
5. Reserva espacio para los buffers y rellena el primero con la cantidad de datos especificada a transferir sobre estos canales.

Fase de Ejecución

1. El cliente inicia la ejecución del DSP.
2. Rellena el búfer de salida con datos de ejemplo.
3. A continuación dirige el buffer por el canal CHNL_ID_OUTPUT y espera hasta volver a obtenerlo. La espera se define por tiempo indefinido.
4. La finalización de la operación anterior indica que el buffer se ha transmitido satisfactoriamente por el enlace físico.
5. Suministra un buffer vacío en el canal CHNL_ID_INPUT y espera hasta obtenerlo. La espera se especifica por tiempo indefinido.
6. Una vez el buffer se ha recogido, su contenido se compara con el buffer enviado, y si no ha habido ningún error, deberían ser iguales.
7. El cliente repite los pasos 3-6 según el número de iteraciones especificado por el usuario.
8. Detiene la ejecución del DSP.

Fase de Finalización

1. El cliente libera los búfers reservados para la transferencia de datos.
2. Elimina los canales CHNL_ID_INPUT y CHNL_ID_OUTPUT.
3. Cierra la *pool*.
4. Se desvincula del DSP y destruye el componente PROC.

EN EL LADO DSP:

Fase de Inicialización

1. La tarea cliente tskLoop se crea en la función main().
2. En el main() se reserva el espacio para la pool y configura el tamaño de los buffers que se usarán
3. Esta tarea crea los canales SIO llamados INPUT_CHANNEL y OUTPUT_CHANNEL para el intercambio de datos.
4. Inicializa los buffers que se usarán para la transferencia de datos.

Fase de Ejecución

1. La tarea señaliza un búfer vacío sobre el canal INPUT_CHANNEL y espera hasta obtenerlo. Esta espera tiene carácter indefinido.
2. A continuación se realiza una FFT y luego se transmite el buffer sobre el canal OUTPUT_CHANNEL y se espera hasta obtenerlo. De nuevo la espera es indefinida.
3. El final de la operación de espera indica que el buffer ha sido transferido a través del enlace físico.
4. Estos pasos se repiten el número de iteraciones especificado en el argumento pasado al ejecutable de la aplicación.

Fase de Finalización

1. La tarea libera los buffers reservados para la transferencia de datos.
2. Elimina los canales SIO INPUT_CHANNEL y OUTPUT_CHANNEL.

CAPÍTULO 3

DESARROLLO DE APLICACIONES

En este capítulo se explican dos aplicaciones que abarcan todos los puntos comentados hasta ahora en este trabajo. Es de especial interés académico el reproductor de radio, puesto que es el ejemplo más representativo de todos los sistemas multimedia que se han presentado.

Por otro lado, el desarrollo de drivers es sin duda un tema imprescindible a la hora de crear un sistema personalizado, puesto que será habitual tener que escribir nuestros propios controladores para los circuitos integrados que deseemos añadir a nuestra placa. Este ejercicio muestra paso a paso, empleando como dispositivo de ejemplo un conversor analógico-digital ADC0831 de un canal, desde la creación de un módulo clásico hasta la interfaz del bus SPI al kernel del sistema.

3.1. Reproductor de radio MP3 por Internet

El objetivo de esta aplicación es la de conectarse a través de Internet a un servidor de música tipo emisora de radio, cuya emisión la realiza en formato MP3 y reproducir el sonido a través del sistema de audio de la placa. Además se ha añadido una simple interfaz gráfica a modo de demostración de los sistemas visual y táctil, desde donde podremos iniciar o detener la reproducción de música.

3.1.1. Emisiones de música sobre protocolo SHOUTcast

El protocolo de streaming de audio Shoutcast
Creado en 1999 por la empresa Nullsoft (creadora del famoso reproductor Winamp), este sistema permitía la difusión por Internet de emisiones de sonido, con lo que en poco tiempo empezaron a surgir estaciones de radio en Internet. Utiliza como protocolo de transporte el HTTP y por su simplicidad, está integrado en múltiples programas de reproducción de audio tales como VLC y Apple iTunes. El protocolo es sencillo y tras conectarse el cliente al servidor de música indicado, lo primero que éste le envía es una cabecera que contiene información básica acerca de la emisión a la que se ha conectado en texto plano. Un ejemplo de cabecera lo podemos ver a continuación:

```
ICY 200 OK
icy-notice1: <BR>This stream requires <a
href="http://www.winamp.com/">Winamp</a><BR>
icy-notice2: Firehose Ultravox/SHOUTcast Relay Server/Linux
v2.6.0<BR>
icy-name: .977 The Hitz Channel
icy-genre: Pop Rock Top 40
icy-url: http://www.977music.com
content-type: audio/mpeg
icy-pub: 1
icy-metaint: 16384
icy-br: 128
```

La primera respuesta consiste en ICY 200 OK, indicando que efectivamente se trata de un servidor Shoutcast y devuelve el código de respuesta 200 indicando que la conexión es correcta. Las siglas ICY tienen un origen histórico: originalmente el protocolo se denominaba I Can Yell (puedo gritar, en inglés), pero por un problema de registro de dominio, se vieron obligados a cambiar el nombre, aunque conservaron los detalles de las cabeceras.

A continuación se reciben una lista de etiquetas con información muy específica. Entre ellas, destacamos las siguientes:

- **icy-notice1**: Primera línea de mensaje de bienvenida. La mayoría de servidores que se han probado para este trabajo han contestado en esta línea que se necesita el reproductor Winamp para poder escuchar esta emisión.

- **icy-notice2**: Segunda línea del mensaje de bienvenida, en el que se aprovecha para facilitar alguna información extra del servidor. En el ejemplo mostrado se detalla la versión y software del servidor que está ejecutándose.

- **icy-name**: Este es el nombre de la emisora que retransmite la música

- **icy-genre**: Género de la música que vamos a escuchar

- **icy-url**: Dirección web de esta emisora, donde podemos encontrar más información acerca de los servicios que ofrece

- **content-type**: Detalle concreto del tipo de contenido que se está emitiendo así como del formato en el que está codificada la música. En el ejemplo mostrado, se trata de una emisora de audio que emite la música con una compresión mpeg, sin especificar versión.

- **icy-pub**: Se trata básicamente de un interruptor que permite al servidor anunciarse en un directorio o no. 1 significa SI mientras 0 significa NO.

- **icy-metaint**: Indica el número de bytes de música que hay entre dos metadatos consecutivos. Los metadatos proporcionan información del título de la canción y autor que estamos escuchando en estos momentos.

- **icy-br**: Informa de la velocidad de bits (bitrate) a la que esta estación emitirá la música codificada. En el caso de que la música se encuentre en formato MP3, generalmente coincidirá que el bitrate aquí indicado también coincide con el bitrate del stream MP3 codificado.

La transmisión de la cabecera finaliza con una línea en blanco con su correspondiente salto de línea y retorno de carro (caracteres 0x0D 0x0A dos veces consecutivos). A partir de este momento, todos los bytes que se reciben consisten en el stream de datos de audio en el formato que nos ha informado anteriormente el servidor.
Para la gestión de la conexión y comunicación con el servidor Shoutcast, se ha creado un conjunto de funciones agrupadas en el módulo de TCP, que se recogen en el fichero tcp.c. A continuación se expone una breve descripción de lo que realizan las funciones más importantes:

- *int tcpConnectServer(char servidor[], int port, char path[])*

Se conecta al puerto del servidor especificado y cuando lo haya conseguido le solicitará la cabecera HTTP de la emisión enviándole una orden similar a la línea siguiente:

GET path/de/la/radio HTTP/1.1

Por ejemplo, al buscar en Internet encontramos un servidor de música en la siguiente dirección web:

http://scfire-mtc-aa06.stream.aol.com:80/stream/1048

la función tcpConnectServer() abrirá una conexión al puerto 80 del servidor scfire-mtc-aa06.stream.aol.com y tras conectar satisfactoriamente con el servidor web general, le enviará la siguiente cabecera HTTP:

GET /stream/1048 HTTP/1.1

Tras esto, el servidor nos enviará la cabecera Shoutcast analizada anteriormente y empezaremos a recibir los datos codificados de música. Por último comentar que esta función devuelve el número de socket abierto para esta conexión TCP/IP y que será este mismo socket el que usemos durante el resto de la emisión para leer los datos transmitidos por el servidor.

- *int tcpMP3Sync(int sock, unsigned int *header);*

Esta función se sincroniza con una cabecera de trama MP3. Para ello, va buscando byte a byte el patrón de sincronización en tramas MP3, que es el siguiente:

- Para versiones MPEG 1 y 2, nos sincronizaremos buscando 12 bits consecutivos a 1
- Para la versión MPEG 2.5, buscaremos 11 bits a valor 1

Una vez encontrada la cabecera de trama MP3, la guardaremos por separado puesto que contiene información acerca del formato de compresión en el que vienen los datos de música.

- *int tcpGetMetadata(int sock,struct emisora_t *emi);*

Esta función lee los metadatos que vienen en el stream de datos que nos proporciona el socket. Se presupone que el puntero de lectura del socket ya está posicionado correctamente sobre el primer byte de los metadatos. El formato de esta información cumple un detalle importante. El primer byte de los metadatos indica el número de bytes del tamaño completo de los metadatos, pero dividido entre 4. Es decir, si el primer byte que leemos es 16, querrá decir que los siguientes 16x4 = 64 bytes son metadatos y no música, por lo que se tienen que extraer del flujo de bits que luego le enviaremos al descodificador para que no oir pitidos ni artefactos extraños en la reproducción del sonido.

Nótese que este primer byte que indica la longitud perfectamente puede adoptar el valor 0, indicando por tanto que realmente no hay metadatos. Esto resulta ser bastante frecuente, puesto que mientras el servidor no cambie de canción, no resulta necesario estar enviando los metadatos tan frecuentemente con el fin de ahorrar bits innecesarios.

- *int tcpGetHTTPResponse(int sock);*

Esta función evalúa la respuesta HTTP del servidor cuando nos conectamos con tcpConnectServer. En condiciones normales, el servidor contestará con el código 200, indicando que todo es correcto. Pero es posible que responda con otros códigos para indicar algún error del momento, como por ejemplo el código 302, que significa que el recurso solicitado se haya temporalmente en otro lugar.

- *void tcpGetHTTPHeader(int sock, struct emisora_t* emi);*

Esta función es la que recoge línea a línea toda la cabecera del servidor cuando nos conectamos. Termina cuando encuentra 2 finales de línea seguidos sin contenido entre ellas, lo que indica en el protocolo HTTP que se ha llegado al final del envío de la cabecera.

- *void tcpCloseConnection(int sock);*

Por último se dispone de una función para finaliza la conexión con el servidor, cuyo misión principal es la de desconectar el socket devuelto por tcpConnectServer() y liberarlo.

3.1.2. Descodificación de MP3

Para este trabajo, se han empleado servidores que retransmitiesen audio en formato MP3. Es por ello que se necesita descodificar esta información para poder obtener las muestras de audio que necesita un conversor digital-analógico para poder reproducir la música en unos altavoces.

Descodificar MP3 no es tarea simple ni corta, y para ello se ha recurrido a una librería gratuita y Open Source, denominada MAD (MPEG Audio Decoder), de la empresa Underbit. Dicha librería se ha compilado para nuestro entorno y se ha empleado como descodificar de los datos en MP3 que se reciben del servidor Shoutcast.

Podemos descargar los paquetes necesarios para compilar la librería MAD directamente de la página web de sus creadores:

http://www.underbit.com/products/mad/

- **LIBRERIA PARA TAGS EN FICHEROS MP3 (AUTOR, TITULO, ETC...)**

ftp://ftp.mars.org/pub/mpeg/libid3tag-0.15.1b.tar.gz
(330 KB)

Que compilaremos de la siguiente forma:

```
> ./configure --prefix=$TOOLCHAIN --host=arm-none-linux-gnueabi
  --enable-fmp=arm
> make
> make install
```

Y nos guardamos libid3tag.so.0 para copiarla al directorio /lib de la placa.

- **LIBRERIA PARA DESCODIFICAR FICHEROS MP3**

ftp://ftp.mars.org/pub/mpeg/libmad-0.15.1b.tar.gz
(490 KB)

Para poder compilar este paquete, hay que editar el fichero *Makefile* tras haber ejecutado el script de configuración y borrar la opción en CDFLAGS que especifica *–fforce-mem*, porque nuestro compilador arm-none-linux-gnueabi-gcc no soporta esta opción.

```
> ./configure --prefix=$TOOLCHAIN --host=arm-none-linux-gnueabi
  --enable-fmp=arm
> make
> make install
```

Ya podemos emplear las funciones que ofrece esta librería. Para ello recordemos que desde nuestro código fuente deberemos incluir las cabeceras de las funciones con la directiva:

```
#include <mad.h>
```

Y a la hora de compilar, deberemos enlazar contra la librería MAD de la siguiente forma:

```
>arm-none-linux-gnueabi-gcc programa.c -o programa -lmad
```

ENTENDIENDO LA LIBRERIA DE DESCODIFICACION MP3 LIBMAD

Se recomienda consultar un ejemplo sencillo incluido en el código fuente con nombre “minimad.c”., en el mismo directorio que el código fuente de la librería. Para compilarlo para la placa escribiremos la siguiente instrucción:

```
>arm-none-linux-gnueabi-gcc minimad.c -o minimad -lmad
```

Para ejecutar este programa y ver su demostración, hay que usarlo de la siguiente forma:

```
>./minimad <entrada.mp3 >salida.wav
```

A continuación se comentan los pasos necesarios para poder descodificar datos MP3 usando la librería MAD:

1. Declarar las estructuras que necesita el descodificador para operar:

```
struct mad_stream Stream;
struct mad_frame Frame;
struct mad_synth Synth;
```

2. Inicializar las estructuras declaradas:

```
mad_stream_init(&Stream);
mad_frame_init(&Frame);
mad_synth_init(&Synth);
```

3. Cargar el búfer de descodificación con nuestros datos:

mad_stream_buffer(&Stream,InputBuffer,ReadSize+Remaining);

Se presupone que InputBuffer se ha llenado con los datos procedentes del socket conectado al servidor, que tiene un tamaño ReadSize y que quedan en el búfer Remaining bytes no descodificados del bucle anterior.

4. Descodificar la cabecera de la siguiente trama MP3 y las muestras de las subbandas

mad_frame_decode(&Frame,&Stream)

5. Sintetizar las muestras de audio PCM con las muestras de las subbandas del paso anterior:

mad_synth_frame(&Synth,&Frame);

Llegados a este punto, las muestras de audio están contenidas dentro de la estructura *mad_synth*, cuya declaración es la siguiente:

```
struct mad_synth {
  /* polyphase filterbank outputs */
  mad_fixed_t filter[2][2][2][16][8];
 /* [ch][eo][peo][s][v] */
  unsigned int phase;   /* current processing phase */
  struct mad_pcm pcm;   /* PCM output */
};
```

Los datos PCM se guardan pues en una estructura del tipo *mad_pcm*, cuya forma se detalla a continuación:

```
struct mad_pcm {
  unsigned int samplerate;    /* sampling frequency (Hz) */
  unsigned short channels;    /* number of channels */
  unsigned short length;      /* number of samples per channel */
  mad_fixed_t samples[2][1152];/* PCM output samples [ch][sample]*/
};
```

Aquí ya vemos que el miembro *samples* de la estructura *mad_pcm* consiste en un array bi-dimensional (para canal izquierdo y derecho) de las muestras que ha descodificado anteriormente. El número 1152 son el número de muestras de la señal de audio que se general al descodificar una trama MP3 completa. Este número es siempre constante.

6. Liberar las estructuras empleadas por el descodificador:

```
mad_synth_finish(&Synth);
mad_frame_finish(&Frame);
mad_stream_finish(&Stream);
```

Es importante liberar todas las variables y estructuras para las que hayamos tenido que reservar memoria, puesto que de lo contrario, el sistema se iría quedando sin memoria y tarde o temprano acabaría por corromperse, resultando en una situación imprevisible y de difícil solución.

ENLAZANDO CON LA INTERFÍCIE GRÁFICA

Las librerías gráficas que se han comentado anteriormente, Qt de Nokia, y más concretamente el entorno de programación gráfico Qt Creator, emplean como lenguaje de programación C++. Por lo tanto, tendremos que mezclar nuestro código en C con programas escritos en C++. Para realizar esto, basta con declarar en C++ que un fichero está escrito en lenguaje C para que el compilador lo entienda. Esto se realiza de la siguiente manera:

```
extern "C" {
    #include "radiomp3.h"
}
```

Para que al presionar un botón en el entorno gráfico se ejecute un código específico, deberemos de configurar la acción desde Qt Creator. Para ello, y tras haber insertado algún botón en la interfaz de usuario (fichero .ui), haremos clic con el botón derecho sobre el botón, y en el menú desplegable elegiremos la opción “Go to slot…”.

En la ventana que aparece a continuación, se muestran los posibles eventos que genera este botón. Nos interesa el evento clicked(), seleccionamos y pinchamos sobre el botón “OK”.

Hecho esto se nos abrirá el editor de código, y nos habrá insertado automáticamente el código necesario, que será similar al siguiente:

```
void MainWindow::on_pushButton_7_clicked()
{
        radio_main();
}
```

Para llamar a nuestro código principal, ya podemos escribir directamente entre las llaves que nos aparecen.

Responsividad de un sistema

El concepto de responsividad hace referencia a la sensación de respuesta que transmite una interfície frente al usuario. Es decir, que cuando interactuamos con el sistema, queremos ver que el sistema está haciendo algo. Esto se consigue con algunos efectos gráficos como por ejemplo juego de sombras al presionar un botón, redibujar un botón de posicionamiento al redefinir su valor o la señal de "*OK*" al marcar un cuadrado de opción tipo "*Checkbox*". Todos estos efectos gráficos contribuyen a una experiencia de usuario más rica y sobre todo indican al usuario que el sistema está haciendo algo solicitado y que no se ha quedado colgado.
Como bajo ningún concepto podemos permitir que la responsividad del sistema disminuya, todas las operaciones que se hagan de cálculo o algorítmica en general, será recomendable realizarlas en un hilo aparte (thread en inglés), para que el hilo principal que controla el aspecto gráfico, no quede interrumpido nunca. Para lanzar el proceso que descodifica:

```
stop_playing = 0;
pthread_create(&audio_decoder,NULL,&MpegAudioDecoder,NULL)
```

Y para esperar a que finalice:

```
stop_playing = 1;
pthread_join(audio_decoder,NULL);
```

3.2. Desarrollo de drivers

3.2.1. Introducción

Un driver o controlador es un programa que sirve de intermediario entre un dispositivo de hardware y el sistema operativo. Su finalidad es la de permitir extraer el máximo de las funcionalidades del dispositivo para el cual ha sido diseñado.

A la hora de crear una plataforma nueva, será habitual tener que escribir nuestros propios drivers para los circuitos integrados que hayamos escogido usar como dispositivos periféricos. En esta sección, se explica cómo se crea un driver básico para Embedded Linux y se desarrolla un ejemplo completo.

En Linux, todos los dispositivos hardware se reflejan en el sistema operativo como si fueran ficheros. Incluso dispositivos virtuales quedan también mapeados en el sistema como otro fichero más. Esta forma de proceder simplifica el modo de trabajar para los programadores, a la vez que ofrece un marco unificado para tratar dispositivos. Esto significa que si deseamos enviar datos a un dispositivo, en principio será suficiente con abrir el fichero y realizar un acceso de escritura sobre él.

El código que ejecuta un driver es un tanto delicado y debe ser ejecutado con precaución, puesto que al estar tratando directamente a nivel de hardware, si hubiera un fallo, podría volver el sistema entero en inestable. Es por eso que el acceso al hardware en Linux lo realiza siempre el sistema operativo por nosotros. Dicho de otro modo, todo el código que ejecuta un driver, lo está ejecutando realmente el kernel del sistema operativo por nosotros. La mayoría de accesos a hardware, registros, memoria y demás queda bloqueada al usuario normal. Estos modos de trabajar es lo que se conoce como espacio de kernel y espacio de usuario. En inglés, se denoniman kernel space y user space, respectivamente.

Kernel Space vs. User Space

Se denomina *kernel space* a aquel espacio de memoria donde se está ejecutando el sistema operativo y sus procesos internos, a la vez que otros programas indispensables, como son los drivers para controlar los dispositivos hardware asociados al sistema. El kernel space tiene acceso a todo el hardware sin restricciones ni limitaciones.

Por contra, el *user space* es un espacio de memoria protegida que es donde el usuario ejecuta sus programas. Si un usuario desea emplear por ejemplo una impresora, el programa editor de textos se está ejecutando en espacio usuario,

pero cuando vaya a imprimir, le enviará la información al driver de la impresora y será éste desde espacio kernel el que se comunicará con la impresora para enviarle la información que debe de imprimir.

Lógicamente, a la hora de escribir drivers para un kernel, se debe de tener en cuenta su versión, puesto que especialmente en el mundo Linux, los cambios son habituales y bastante rápidos. Concretamente, en la última versión 2.6.x del kernel, se ha cambiado radicalmente el modelo de drivers, lo que se conoce como Linux Device Model, y han variado muchas funciones del sistema típicamente usadas en el desarrollo de drivers.

Es por ello que cuando desarrollamos un driver, el proceso de compilación pasa por compilar contra el mismo kernel para el que estamos desarrollando el driver. En el árbol de directorios del kernel encontramos un directorio bajo el nombre de 'include', que es donde podemos encontrar todos los ficheros .h con las definiciones de las funciones disponibles en el espacio kernel.

No olvidemos que el kernel tiene sus propias funciones y que no depende de la librería C estándard que habitualmente usan los programadores para escribir programas pensados para espacio usuario. El kernel depende únicamente de sus propias funciones. Por suerte para el programador, la mayoría de funciones que usaremos para desarrollar drivers se llaman prácticamente igual que su versión homóloga en la librería C estándard. Para diferenciarlas, suelen anteponer o añadir al final la letra 'K' al nombre para distinguir que esa función particular está disponible para el kernel, como por ejemplo *printf* y *printk* o *malloc* y *kmalloc.*

3.2.2. Compilación de un driver

Para poder compilar con comodidad, nos definiremos un fichero *Makefile* el directorio donde tengamos el código fuente de nuestro driver, que nos ayudará a asegurarnos de que nuestros drivers compilen contra el kernel que nos interese.

Por ejemplo, examinemos el siguiente fichero *Makefile*:

```
obj-m := mydriver.o

KERNELDIR := $(HOME)/projects/linux-2.6.28-omap
PWD := $(shell pwd)

default:
	$(MAKE) -C $(KERNELDIR) SUBDIRS=$(PWD) modules
```

La opción –C obliga a *make* a cambiar al directorio especificado como primera acción, y desde el nuevo directorio, continuar la ejecución. De esta forma nos aseguramos de que el compilador encontrará todos los ficheros de cabeceras (.h) en el directorio que nos interesa exactamente.

Dado que en el directorio del código fuente del kernel ya tenemos un fichero *Makefile* configurado para que utilice los compiladores cruzados arm-none-linux-gnueabi-XXX, no necesitaremos en nuestro *Makefile* volver a especificarlo, puesto que ya *make* recoge esta configuración por defecto.

La primera sentencia indica qué ficheros intervienen para la compilación del driver. Se entiende que debe existir un fichero denominado igual que el declarado, pero con extensión .C, que en este caso debería ser *mydriver.c*. Si por alguna razón nuestro driver se compone de 2 o más ficheros .C, la declaración correcta sería entonces:

```
obj-m := mydriver.o
module-objs := fichero1.o fichero2.o

KERNELDIR := $(HOME)/projects/linux-2.6.28-omap
PWD := $(shell pwd)

default:
```

Un detalle importante es que no debe haber espacios en blanco antes de $(MAKE), sino un único carácter de tabulador. En caso contrario, *make* no reconoce la línea y la omite.

3.2.3. Tipos de drivers en Linux

Existen dos tipos principales de dispositivos en todos los sistemas, denominados dispositivos de caracteres y dispositivos de bloque. En inglés, se conocen como *character devices* y *block devices*. Los *character devices* son aquellos dispositivos que no necesitan ningún tipo de buffering, por lo que permiten un acceso directo. En cambio, los *block devices* son aquellos que necesitan algún tipo de memoria intermedia o caché para su acceso. Un disco duro o una memoria RAM son ejemplos de dispositivos de bloque.

Por su naturaleza, los *block devices* deben de soportar acceso aleatorio, es decir, poder acceder directamente a cualquier parte de su estructura organizativa sin tener que antes leer toda la anterior. Los dispositivos de caracteres no necesitan cumplir con este requisito. Por ejemplo, si se desea montar un sistema de ficheros, éste debe de obligatoriamente estar implementado sobre un *block device*.

Los accesos de lectura y escritura a un dispositivo de caracteres se realiza con las funciones *read()* y *write()* respectivamente. Estas funciones tienen la particularidad que no retornan hasta que la operación no se ha completado. En cambio, los dispositivos de bloque ni siquiera disponen de funciones similares a *read()* y *write()*, sino que emplean una función que históricamente se ha llamado *rutina de estrategia*. Esta función lo que hace es ir a consultar un buffer intermedio, o memoria caché según el bloque solicitado, de la que recoger los datos. Si allí no los encuentra, entonces realiza el acceso de lectura real al dispositivo. Si lo que se desea es escribir en el dispositivo, entonces comprueba previamente si el buffer de escritura no está lleno. Los accesos a este dispositivo pueden ser asíncronos. Por ejemplo, una función utilizada típicamente, *breada(),* planifica lecturas del dispositivo para cargar en el buffer intermedio en un proceso en segundo plano con la esperanza de que dichos datos se necesiten más adelante.

En este apartado nos basaremos en *char devices* para llevar a cabo los ejemplos de como desarrollar un driver básico y más adelante un driver completo para un conversor analógico-digital.

3.2.4. Ejemplo de un módulo básico

A continuación se explican los pasos básicos y que serán habitualmente comunes en la mayoría de drivers que necesitemos desarrollar. En primer lugar empezaremos a ver la carga y descarga de un módulo genérico en el kernel de Linux. Posteriormente veremos como interactuar con el sistema de ficheros para que los usuarios puedan abrir el dispositivo como si fuera un fichero más y poder realizar accesos de lectura y escritura.

Cargar y descargar un módulo

El siguiente código de ejemplo muestra la mínima expresión que puede adoptar un módulo que queramos desarrollar para insertar en el kernel del sistema.

```
#include <linux/init.h>
#include <linux/module.h>
#include <linux/kernel.h>

MODULE_LICENSE("Dual BSD/GPL");

static int hello_init(void) {
  printk("<1> Hello world!\n");
  return 0;
}
static void hello_exit(void) {
  printk("<1> Bye, cruel world\n");
}
module_init(hello_init);
module_exit(hello_exit)
```

La macro *module_init()* le indica al kernel qué función debe de ejecutar cuando cargue este módulo. Por contra, la macro *module_exit()* le indica al kernel qué función ejecutar al descargar el módulo de memoria. Si estamos cargando el módulo desde el espacio de usuario, normalmente lo haremos con la instrucción *insmod* (INSert MODule), y la descarga del módulo lo haremos con la instrucción *rmmod* (ReMove MODule).

Acceso por lectura y escritura

Para que el usuario pueda interactuar con el driver, se debe de crear un fichero especial al cual, en función de la naturaleza del dispositivo, concederemos funciones para hacer lecturas y escrituras. En Linux, los dispositivos quedan mapeados a un fichero bajo el directorio /dev (del inglés devices, dispositivos). En nuestro driver, deberemos de en primer lugar crear y registrar el dispositivo bajo el árbol /dev, y en según lugar, asignar qué operaciones se pueden realizar sobre ese fichero.

Para ello disponemos de la función:

```
int register_chrdev(    unsigned int major,
                        const char *name,
                        struct file_operations *fops);
```

Sus argumentos son:

- **unsigned int major**: indica el número MAJOR del dispositivo que vamos a registrar. El MAJOR es un número que indica la categoría o grupo al que pertenece el dispositivo. Por ejemplo, el número 6 representa impresoras que imprimen por el puerto paralelo, mientras que el 188 representa conversores de puerto USB a puerto serie RS232, como el usado en este trabajo para conectar con la placa. Así mismo existe el concepto de MINOR, que es una subclasificación del MAJOR. Todos los dispositivos que registremos deben de asignarse a un MAJOR y un MINOR. Si no estamos seguros del MINOR, existe la posibilidad de hacer esto de forma automática y dinámica. La lista completa de MAJOR i MINOR, que representa la máxima autoridad y debe respetarse, la podemos consultar en el directorio de documentación del propio código fuente: $HOME/projects/linux-2.6.28-omap/Documentation/devices.txt
- **const char *name**: será el nombre que queramos que aparezca bajo el directorio /dev
- **struct file_operations *fops**: puntero a una estructura que indica el tipo de operaciones que el driver va a soportar a nivel de fichero, es decir, si permitimos escrituras, lecturas, cambios de puntero, etc…

La estructura *fops* es muy importante, puesto que es la que realmente proporciona los accesos de cara al usuario. Un ejemplo de cómo rellenar y usar esta estructura lo podemos ver a continuación:

```
struct file_operations memory_fops = {
  read: funcion_read,
  write: funcion_write,
  open: funcion_open,
  release: funcion_release
};
```

Esto es una forma peculiar de, a la vez que se declara una variable (en este caso se llama *memory_fops*) aprovechar para rellenar los miembros de la propia estructura. Este estilo de declaración es muy típico dentro del kernel de Linux y lo podremos ver en muchos otros ficheros dentro de su código fuente.

Las funciones básicas que se muestran en el ejemplo anterior son las siguientes:

- **funcion_read**: representa la función que queremos que se ejecute cuando el usuario desde su aplicación realiza un acceso de lectura mediante la función de espacio de usuario *read()*
- **funcion_write***:* similar a la función anterior, pero esta vez el acceso es por escritura, y en espacio de usuario se emplea la función *write()*
- **función_open**: la función que queremos que se ejecute cuando desde el espacio usuario abren el fichero con la función *open()*
- **funcion_release**: es la función contraria a *open()* desde el espacio de usuario, *close()*, que llama el usuario cuando no va a usar más el dispositivo

Las funciones que especifiquemos en la estructura *fops* deben de cumplir con el siguiente prototipo que nos impone Linux:

```
int funcion_open(struct inode *inode, struct file *filp);

int funcion_release(struct inode *inode, struct file *filp);

ssize_t funcion_read(struct file *filp, char *buf, size_t
count, loff_t *f_pos);

ssize_t funcion_write(struct file *filp, char *buf, size_t
count, loff_t *f_pos);
```

3.2.5. Desarrollo de un driver completo: ADC0831

La gama ADC0803x son una serie de conversores analógico – digitales ofrecidos por varios fabricantes, con las siguientes características:

ADC0831 : 1 canal
ADC0832 : 2 canales
ADC0834 : 4 canales
ADC0838 : 8 canales

En nuestro caso, disponemos de una unidad ADC0831. Estos conversores se conectan al microprocesador a través de un bus serie denominado Microwire. Este bus es similar al SPI, aunque al ser una especificación anterior, no permite tanta flexibilidad como el SPI. En concreto, la mayor diferencia consiste en que SPI permite configurar la fase y la polaridad de la señal de CLOCK, mientras que en Microwire esto no es posible. Aparte de estos detalles, los buses son completamente compatibles y de hecho emplearemos el bus SPI integrado en el OMAP3530 para leer del dispositivo. En la figura siguiente vemos el patillaje detallado del ADC0831:

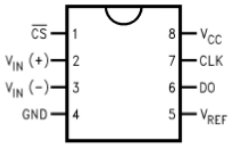
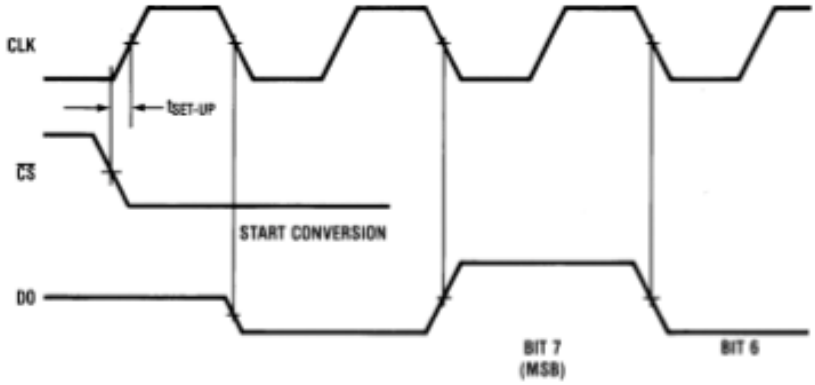


Figura 23: Pinout del ADC0831

Para indicarle al conversor ADC que debe de iniciar una conversión y devolvernos el valor digitalizado, basta con realizar un acceso de lectura por su interfície serie, como muestra la figura:



Como se puede ver, la conversión se inicia con el siguiente flanco de bajada tras la activación del Chip Select. El orden en el que el dispositivo nos devuelve el valor digitalizado es el bit más significativo primero (MSB), que corresponde con el bit 7.

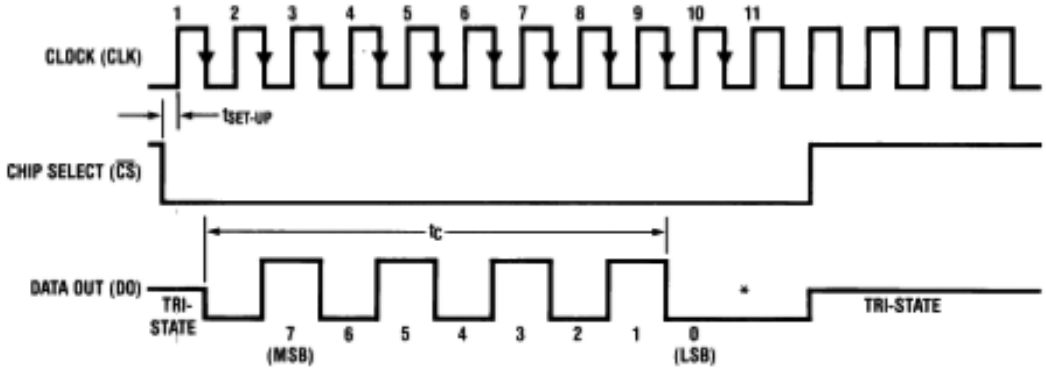


Figura 24: Diagrama de timing para el ADC0831

A nivel eléctrico, necesitamos poder conectar el dispositivo a la placa DevKit8000. Esto es posible gracias a un conector de expansión de 40 pins que hay previsto en la propia placa y que sirve para ampliar el número de periféricos que podemos conectar al OMAP3530. En la figura siguiente vemos el patillaje de dicho conector:

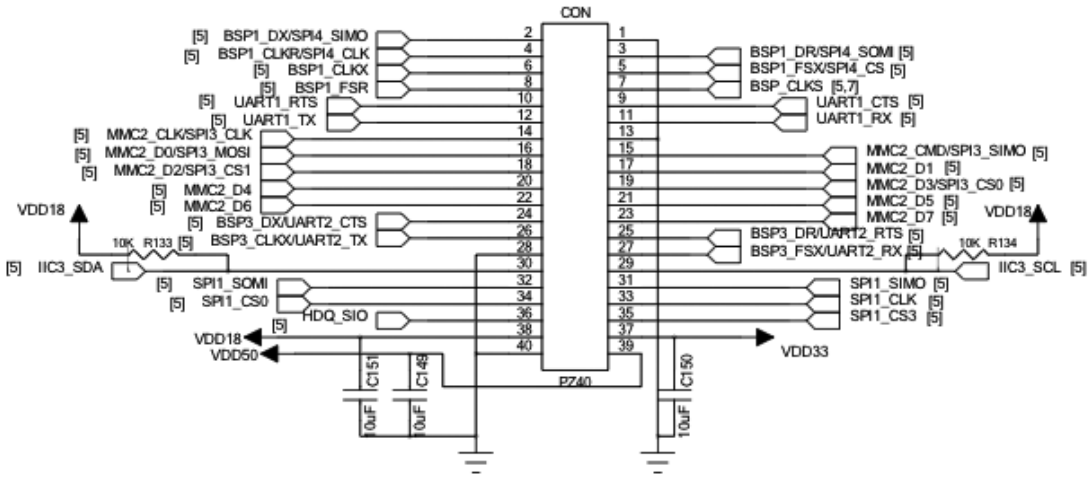


Figura 25: Detalle del conector de 40 pins de la placa DevKit8000

Para nuestro ejemplo, aprovecharemos el puerto SPI1. Si nos fijamos en el esquema anterior, vemos que para dicho bus disponemos de dos señales Chip Select, la SPI1_CS0 y SPI1_CS3. Esto significa que podemos conectar hasta 2 dispositivos al bus SPI1. En este caso elegimos el SPI1_CS0, aunque podríamos haber empleado el SPI1_CS3 por igual. Por tanto, la conexión del ADC0831 queda de la siguiente forma:

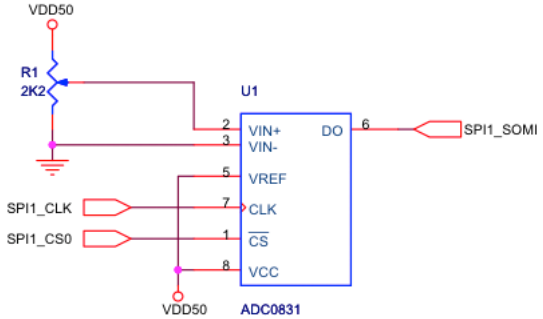


Figura 26: Esquema de la conexión del ADC0831 a la placa

En el siguiente fichero C dentro del kernel de Linux, vemos cómo se declaran los dispositivos que tenemos conectados a nuestra placa a través de los puertos SPI:

/home/ramonvp/projects/linux-2.6.28-omap/arch/arm/mach-omap2/board-omap3devkit8000.c

```
struct spi_board_info omap3devkit9100_spi_board_info[] = {
[0] = {
      .modalias         = "ads7846",
      .bus_num          = 2,
      .chip_select      = 0,
      .max_speed_hz     = 1500000,
      .controller_data  = &ads7846_mcspi_config,
      .irq              = OMAP_GPIO_IRQ(OMAP3_BEAGLE_TS_GPIO),
      .platform_data    = &ads7846_conf,
},
};
```

En el ejemplo mostrado, se puede ver que el único dispositivo conectado al bus SPI es el conversor ads7846, que es el controlador de la pantalla táctil. También puede verse que dicho chip se encuentra en el bus número 2 (SPI2) y que se activa mediante el Chip Select 0 (CS0). La velocidad máxima del bus SPI queda limitada en este caso a 1500 kbps.
Para nuestro propósito, deberemos de configurar aquí el nuevo chip que vamos a conectar al sistema. Si nos fijamos bien, *omap3devkit9100_spi_board_info[]* es un array de tipo *struct spi_board_info* de tamaño 1. Pues bien, únicamente necesitamos añadir 1 elemento más a esta lista de estructuras, que podemos hacer de la forma siguiente:

```
struct spi_board_info omap3devkit9100_spi_board_info[] = {
      {
            .modalias         = "ads7846",
            .bus_num          = 2,
            .chip_select      = 0,
            .max_speed_hz     = 1500000,
            .controller_data  = &ads7846_mcspi_config,
            .irq              = OMAP_GPIO_IRQ(OMAP3_BEAGLE_TS_GPIO),
            .platform_data    = &ads7846_conf,
      },
      {
            .modalias         = "adc0831",
            .bus_num          = 1,
            .chip_select      = 0,
            .max_speed_hz     = 250000,
      },
};
```

En nuestro caso, como se puede ver en la figura 26, hemos conectado el conversor analógico al puerto SPI1, y concretamente al Chip Select 0 (CS0). Hemos especificado una velocidad de datos de 250 kbps, que es el dato que proporciona como valor límite el *datasheet* del producto.

El nombre suministrado en el campo *modalias*, que en este caso es “adc0831” es importante, puesto que a la hora de cargar el driver correspondiente, el kernel buscará por este nombre entre los drivers disponibles con los que lo hayamos compilado.

Por tanto, empezaremos por crear el fichero del driver con el mismo nombre (adc0831.c) y en una ubicación adecuada dentro del árbol del kernel. Dado que este conversor es de propósito general, lo ubicaremos en:

$HOME/projects/linux-2.6.28-omap/drivers/input

Para que al compilar el kernel, se compile también nuestro driver y quede integrado en el propio kernel, deberemos de agregarlo al fichero Makefile correspondiente, que se halla en el mismo directorio especificado antes. En concreto, lo abriremos e incluiremos nuestro driver de la forma siguiente:

```
#
# Makefile for the input core drivers.
#

# Each configuration option enables a list of files.

obj-$(CONFIG_INPUT)           += input-core.o
input-core-objs := input.o ff-core.o adc0831.o
```

Hecho esto, ya estamos en condiciones de empezar a desarrollar el driver en sí. Suponiendo que ya tenemos el fichero “adc0831.c” en el directorio anterior, lo abriremos con un editor de textos y empezaremos a rellenarlo por secciones según se explica a continuación.

Lo primero que necesitamos es una estructura de tipo *spi_driver* que recoge información general acerca de un driver para el bus SPI y que ya está definida en el kernel de Linux. Por ejemplo podría ser así:

```
static struct spi_driver adc0831_spi = {
      .driver = {
            .name =     "adc0831",
            .owner =    THIS_MODULE,
      },
      .probe =    adc0831_probe,
      .remove =   __devexit_p(adc0831_remove),
};
```

De esta estructura, la parte más importante es el puntero a la función que realizará la función *probe*. La función *probe* en un driver se ejecuta cuando se carga por primera vez el driver y se desea comprobar que el dispositivo físicamente no sólo se haya presente sino que además está funcionando correctamente a nivel de hardware. De ahí el nombre. La función *remove*

cumple la función de desconexión del dispositivo justo antes de que lo desconecten físicamente, si esto fuera posible.

Acceso de lectura al dispositivo

Para poder ordenar el conversor que nos devuelva el valor de su última conversión, deberemos realizar un acceso de lectura, lo que automáticamente ya generará las señales eléctricas en el bus SPI necesarias como se ha visto anteriormente. A nivel de driver, se necesitan unas estructuras propias del bus SPI. Una función que nos serviría para leer sería la siguiente:

```
int spidev_sync_read(struct spidev_data *spidev, size_t len)
{
      int status;
      struct spi_transfer transfer = {
            .rx_buf     = spidev->buffer,
            .len        = len,
            };
      struct spi_message m;

      spi_message_init(&m);
      spi_message_add_tail(&transfer, &m);

      status = spi_sync(spidev->spi, message);
      return status;
}
```

Es decir, se crea una estructura que representa la transferencia de datos que se va a producir entre microprocesador y periférico. En el ejemplo se denomina *transfer* , y contiene información del dispositivo del cual tenemos que leer y cuántos bytes queremos obtener, esto último lo indica el campo *len*. A continuación se crea e inicializa un mensaje *m* de tipo *struct spi_message*, que representa el mensaje que se quiere enviar. Es posible realizar varias transferencias en el mismo mensaje, y es por eso que tenemos que añadir a la cola la transferencia de lectura que queremos hacer en este caso. Esto se consigue con la función *spi_message_add_tail()*. Finalmente se llama a la función del kernel *spi_sync*, que es la que realiza la transferencia físicamente. En este ejemplo no se ha tenido en consideración posibles problemas de concurrencia, es decir, si varios procesos intentan acceder simultáneamente al dispositivo. Para solventar esto, se podrían emplear mutex y spinlocks, mecanismos ambos que incluye el sistema operativo para permitir accesos concurrentes.

La función que se ejecutaría cuando el usuario realiza un read desde el espacio de usuario sería algo similar a la siguiente:

```
ssize_t adc0831_read(struct file *filp, char __user *buf, size_t
count, loff_t *f_pos)
{
      struct spidev_data      *spidev;
      ssize_t                 status = 0;
      unsigned long     missing;

      spidev = filp->private_data;

      status = spidev_sync_read(spidev, count);
      if (status > 0) {

            missing = copy_to_user(buf, spidev->buffer, status);
            if (missing == status)
                  status = -EFAULT;
            else
                  status = status - missing;
      }

      return status;
}
```

Si nos fijamos en la función anterior, vemos que realmente no hay ninguna referencia al dispositivo spi en los parámetros que recibimos de la función *adc0831_read().* Esto es habitual, y es por eso que el parámetro filp, que es de tipo estructura *struct file* definida en <linux/fs.h>, tiene reservado un campo para el usuario de tipo puntero genérico. El campo en cuestión es void *private_data, y es lo que permite al programador la flexibilidad de guardar ahí un puntero a la información que necesite para el funcionamiento del driver.

En este ejemplo, se ha creado una estructura de datos privada que representa el dispositivo que estamos tratando:

```
struct spidev_data {
      dev_t             devt;
      spinlock_t        spi_lock;
      struct spi_device *spi;
      struct list_head  device_entry;

      /* buffer is NULL unless this device is open (users > 0) */
      struct mutex            buf_lock;
      unsigned          users;
      u8                *buffer;
};
```

De esta estructura podemos destacar algunos campos:

- struct spi_device *spi: puntero al dispositivo spi que nos interesa
- spinlock_t spi_lock: spinlock para acceder al dispositivo
- dev_t devt: identificador de dispositivo, es un unsigned 32
- Para el caso que se esté compartiendo el dispositivo entre varios usuarios que tengan el fichero abierto, se añade un *mutex* y se lleva la cuenta de usuarios en *users*. *Buffer* se emplea para guardar los datos.

Por lo tanto, para cualquier función del driver, donde el kernel nos esté pasando un puntero al fichero abierto, de ahí recogeremos el puntero al dispositivo SPI que queramos acceder.

Cuando el usuario desee finalizar el uso del dispositivo, deberá cerrar el acceso al fichero mediante la función *close()* en espacio de usuario. Esto hará que se ejecute en el driver la función release(), donde se comprobará si por ejemplo ya no quedan más usuarios que tengan el dispositivo abierto, y si fuera el caso se procedería a liberar de memoria el buffer reservado, así como la propia estructura de datos que nos pasamos entre función y función mediante el struct file. En nuestro ejemplo no es necesario, dado que no van a acceder más usuarios que la aplicación que programemos de ejemplo con fines demostrativos.

CAPÍTULO 4
RESULTADOS

Para evaluar la carga de trabajo de una CPU, Linux lleva integrado unas utilidades que permiten conocer el estado de trabajo de cada núcleo, si hubiera más de uno.

La utilidad más clásica es el comando “top”, que permite ver a tiempo real el estado de carga del sistema. De hecho, muchas de las utilidades gráficas que existen para escritorio en entornos Linux, utilizan la información que proporciona esta herramienta para mostrar la carga.

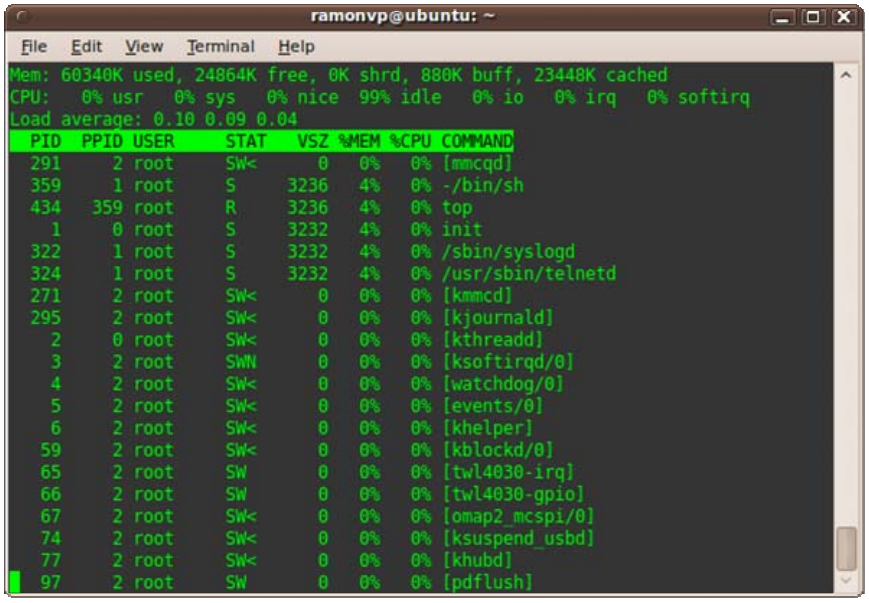


Figura 27: Información de carga que proporciona top

La información que proporciona este comando es muy completa, puesto que además de la carga de la CPU, también indica la cantidad de memoria libre, usada, y la proporción de esta memoria que los procesos están consumiendo de forma individual.

Esta información puede sernos útil para evaluar si nuestra aplicación está consumiendo demasiados recursos del sistema, o por el contrario, está permitiendo que otros procesos que se están ejecutando en el sistema tengan acceso a la CPU. El inconveniente de esta utilidad es que no se comunica con el DSP, con lo que no nos sirve para poder medir la carga del DSP.

Para poder evaluar este dato, Texas Instruments pone a nuestra disposición una utilidad a modo de demostración que sí nos va a indicar la carga del DSP. Para hacer esta medida, se emplea la librería anteriormente descrita, la DSPLINK.

4.1. Carga de la CPU sin usar el DSP

Como prueba de carga para la CPU, le pedimos la reproducción de un vídeo con sonido, sin ayuda del DSP. El programa para su reproducción es el reproductor mplayer, que ya viene pre-instalado en el sistema por el fabricante de la placa DevKit8000. El resultado es que la CPU es, naturalmente, capaz de descodificar por sí misma el vídeo y reproducirlo en la pantalla TFT.

No obstante, el grado de ocupación de la CPU se puede considerar de medio/intenso, puesto que según las medidas, llega a valores cercanos al 40% de utilización. Aún queda mucho margen para que puedan convivir otras aplicaciones en el sistema, pero si pensamos que todo esto lo está gastando únicamente el proceso de reproducción de vídeo, el valor obtenido es bastante alto.

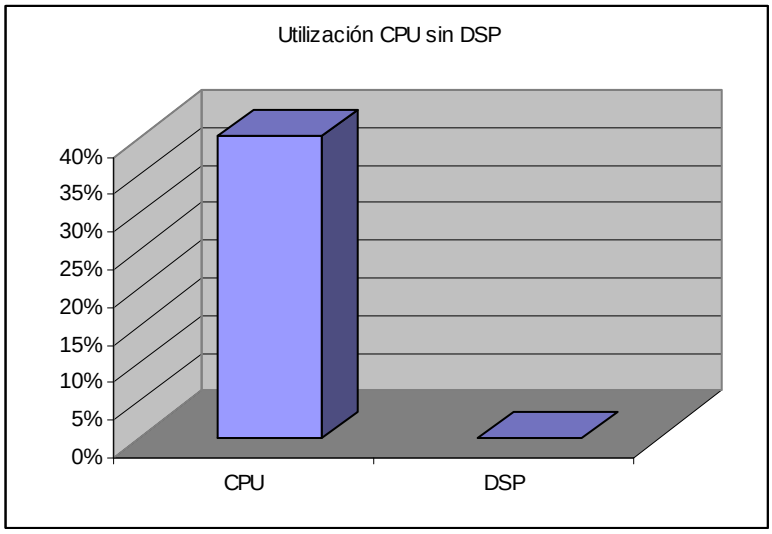


Figura 28: Utilización CPU sin DSP

El sistema a cambio pone en modo espera al DSP para que el consumo de energía sea el mínimo. Por tanto, a la hora de elegir un procesador para el diseño de un determinado sistema embedded, sería conveniente hacer una estimación de la carga que éste va a soportar con todos los procesos funcionando a la vez. Dicho grado de utilización debe de dejar un pequeño margen a la CPU, pues probablemente una utilización al 100% no deje lugar a error alguno. Si se viera que la carga de la CPU supera más del 80%, se debería de replantear el uso de un procesador más potente, incorporar un procesador tipo DSP de ayuda, o simplemente optimizar mucho el código. No siempre es posible elegir entre las 3 opciones.

4.2. Carga de la CPU usando el DSP

En cambio, si permitimos al DSP realizar las tareas de descodificación del mismo vídeo, con las herramientas antes mencionadas de Texas Instruments, el test arroja unos resultados como los siguientes:

```
Decode ARM Load: 12% DSP Load: 92% Display Type: 480P Video Codec:
MPEG4 SP Video fps: 46 fps Video bit rate: 2308 kbps Video
resolution: 720x480 Sound codec: AAC Sound bit rate: 64 kbps
Sampling freq: 48 KHz samp rate Time: 00:00:20
```

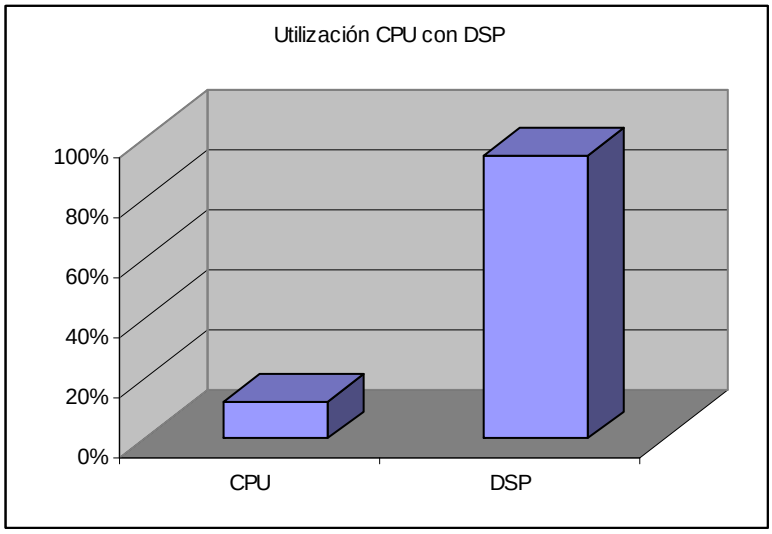


Figura 29: Utilización de CPU y DSP

Donde ahora claramente se está aprovechando al máximo las capacidades del DSP, y se deja libre a la CPU para realizar otras tareas.

CAPÍTULO 5
CONCLUSIONES

En este trabajo se ha evaluado un sistema embedded multimedia completo, que consistía en una placa de desarrollo DevKit8000 del fabricante Embest. Dicha placa incorpora un microcontrolador de la familia Omap3, concretamente el Omap3530, que pertenece a los microcontroladores del tipo SoC, o System in-a-Chip, en el que se pretende incluir la gran mayoría de dispositivos necesarios para formar un ordenador completo en un solo encapsulado.

El estudio de esta placa ha venido motivado por el cambio en la placa que se utilizaba en el laboratorio de la asignatura SETR de esta escuela. Dicha placa, basada en un microcontrolador NS7520, con arquitectura ARM7TDMI, carecía de muchos de los periféricos necesarios para el diseño de un sistema multimedia. Este salto cualitativo entre el modelo antiguo y el nuevo va a permitir a los futuros estudiantes de esta asignatura el aprendizaje sobre un sistema moderno y mucho más versátil, que ofrece más capacidad para probar una amplia variedad de subsistemas.

El objetivo de este trabajo era documentar las capacidades de esta placa, así como la elaboración de guías paso a paso que puedan servir de referencia en el futuro. Con el material aquí expuesto, se elaborará un manual para las prácticas de dicha asignatura, en la que además de añadirse las soluciones al uso de los dispositivos, se plantearán problemas que los alumnos deberán resolver.

Se ha comprobado, como por un lado, el sistema es sencillo de programar y de personalizar, al utilizar un kernel Linux estándar, aunque adaptado a esta plataforma. Y por otro lado el sistema resulta práctico en términos de desarrollo puesto que incorpora todo lo necesario para la construcción de un sistema multimedia. Empezando por su gestión inicial, en la que sólo necesitaremos conectar un puerto serie estándar para empezar a trabajar con esta placa.

Se ha visto la importancia de un gestor de arranque, puesto que es el responsable de arrancar la placa y el que nos va a permitir llevar a cabo las actualizaciones de las diferentes partes del sistema, entre ellas el kernel de Linux (o cualquier otro sistema operativo que deseemos utilizar) y el sistema de ficheros, que por ende será el que incluya las aplicaciones que hayamos desarrollado para el usuario final.

Sin duda, un aspecto clave de la placa DevKit8000 es su interfaz gráfica en combinación con el panel táctil. Esta unión da lugar a muchas posibilidades de interfície usuario-máquina, también conocido por sus siglas HMI (Human-Machine Interface). Una interfaz táctil permite crear botones y controles específicos para cada aplicación, puesto que cada aplicación es diferente y necesita los suyos propios. El cambio de una interfaz basada en botones mecánicos a la eliminación total y el posterior diseño táctil es sin lugar a dudas un salto cualitativo en cualquier producto que se desee evolucionar. Para ello

se han visto las herramientas que hacen esto posible con un esfuerzo mínimo y sobretodo, gratuitamente, al tratarse de herramientas de libre distribución. Muchas empresas que desarrollan productos basados en paneles táctiles verán con buenos ojos el que existan librerías de libre uso, al no haber de invertir en licencias ni pagar royalties.

El apartado del audio, como se ha comentado, viene separado en un circuito integrado externo al microcontrolador principal. No obstante, la unión entre los dos es muy buena y sencilla de utilizar, en parte porque el chip del audio se diseñó precisamente como acompañante al OMAP3530. Además, el uso de la librería ALSA simplifica mucho la reproducción de sonido, al realizar una abstracción de la capa de hardware. Esto hace que el programador no deba preocuparse por los detalles del códec de audio subyacente y pueda concentrarse en la calidad del sonido que desee reproducir.

La parte más importante de este trabajo se ha centrado en el desarrollo de dos aplicaciones que pudieran servir de muestra de todos los sistemas anteriormente comentados. El primero de ellos, el reproductor de radio MP3, quizá sea el más representativo, al tratarse de un entorno gráfico que permite reproducir música proveniente de un servidor de Internet. Se ha explicado el protocolo Shoutcast y en el código fuente de ejemplo puede consultarse los detalles exactos de la implementación y de la secuenciación del stream de bits y de cómo se pasan éstos al descodificador de MP3 para obtener audio digital. El uso de la librería MAD para descodificar MP3 es sin duda un ejemplo importante a nivel académico, pues va a permitir a los futuros estudiantes realizar prácticas con audio con una librería simple de usar, a la vez que gratuita y open-source. El tener el código fuente disponible siempre ayuda a un estudio en mayor profundidad de la herramienta con la que se está trabajando.

El ejemplo de la radio MP3 también aporta al estudiante la visión de cómo se integra por un lado el aspecto gráfico de un programa y cómo se mezcla con su parte funcional por otro. Concretamente, en el entorno Qt Creator se programa con lenguaje C++, mientras que la descodificación y reproducción de audio se realiza en lenguaje C. Poder mezclar ambos lenguajes bajo una misma herramienta es una baza adicional que gana el estudiante, al permitir combinar lo mejor de ambos mundos según lo que se pretenda realizar.

La otra gran aplicación que se ha desarrollado es la guía completa de cómo desarrollar módulos y drivers para Linux, incluyendo un ejemplo completo con un conversor analógico – digital, el ADC0831, que permite ver los mecanismos con los que Linux interacciona con su hardware, en este caso el bus SPI, y cómo el programador debe suministrar las conexiones necesarias para el espacio usuario a través de los ficheros de dispositivos que se recogen en el directorio /dev.

Desarrollar controladores es una tarea por lo general complicada, puesto que un fallo en el driver es la causa principal por la que un sistema operativo puede fallar y volverse inestable. La conexión de un nuevo dispositivo al sistema implica un estudio detallado de las interfaces que pone el kernel de Linux a disposición del programador. Una fuente de documentación imprescindible para el desarrollador de drivers es precisamente el propio código fuente del kernel. Se dispone también de un directorio de documentación, en la que existen algunos textos que cubren temas que pueden servirnos de ayuda en nuestra tarea.

Por último se ha incluido toda la temática relacionada con el procesador digital de señal (DSP) integrado en el propio OMAP3530. Inicialmente se deben de instalar muchas utilidades y librerías, y compilar muchas de ellas para lograr obtener lo que son los módulos que se cargarán en el kernel para permitir la comunicación entre procesador principal (GPP) y el DSP. La tarea es larga, pero permite ver las aplicaciones desde otro punto de vista: ahora ya no se ejecuta todo en un mismo procesador, sino que contamos con la ayuda de un segundo procesador para realizar los cálculos y algoritmos de procesado más repetitivos, que suelen ser los que abarcan la mayor parte el tiempo de trabajo de la CPU. Por tanto, el DSP demuestra según los índices de carga y utilización mostrados, que libera a la CPU de mucho trabajo para dejarla libre para otras tareas. Dichas tareas pueden ser algo tan simple como una comunicación por la red, o bien la responsividad de la interfaz visual, cosas ambas que un programador no debe de descuidar si lo que pretende es obtener un sistema embedded multimedia final de calidad.

En definitiva, la placa DevKit8000 es una buena plataforma donde poder aprender a programar un sistema embedded, con todos sus detalles de implementación y periféricos, y la capacidad de ampliar aún más los dispositivos conectados. El hecho de que venga un kernel de Linux preinstalado y un mecanismo basado en tarjeta SD para poder cargar nuevos sistemas operativos, ofrece al programador y futuros estudiantes mucho juego a la hora de probar en el laboratorio.

BIBLIOGRAFIA

YAGHMOUR, Karim. *Building Embedded Linux Systems*. 1ª ed. O'Reilly & Associates. 2003.

CORBET, Jonathan. RUBINI, Alessandro., and KROAH-HARTMAN, Greg. *Linux Device Drivers*. 3ª Ed. O'Reilly & Associates. 2005.

VENKATESWARAN, Sreekrishnan. *Essential Linux Device Drivers*. Prentice Hall. 2008.

BAKER, Bonnie C. and FANG, Wendy, Texas Instruments, Inc. *Powering resistive touch screens efficiently*. Planet Analog Magazine. May 28, 2007

MATTHEW, Neil and STONES, Richard Stones. *Beginning Linux Programming.* 4ª Ed. Wiley Publishing Inc. 2008

BOVET, Daniel P. and CESATI, Marco. *Understanding the Linux Kernel.* 1ª Ed. O'Reilly & Associates. 2000

Texas Instruments, *TMS320C64x+ DSP Little-Endian DSP Library Programmer's Reference*, Marzo 2006, Edición digital PDF, http://focus.ti.com/general/docs/lit/getliterature.tsp?literatureNumber=sprueb8b&fileType=pdf

BLUM, Richard. *Linux Command Line and Shell Scripting Bible.* 1ª Ed. Wiley Publishing, Inc. 2008

ANEXO I. MENSAJES DE ERROR AL CARGAR LINUX

- **ERROR DE REMOUNT ROOTFS:**

Hay que comentar la primera línea de /etc/fstab que dice:

#rootfs / auto defaults 1 1

- **DESACTIVAR EL DHCP (IP AUTOMATICA) DE ETH0:**

Nos vamos al fichero */etc/network/interfaces*

Y modificamos la sección correspondiente para que quede así:

 # Wired or wireless interfaces
#auto eth0
iface eth0 inet dhcp
iface eth1 inet dhcp

- **AÑADIR UN DNS PARA QUE PODAMOS COMUNICAR CON NOMBRES DE INTERNET:**

- Abrir el fichero /etc/resolv.conf y añadir la siguiente línea:
nameserver 195.235.113.3
O cualquiera otra IP de algún servidor DNS de vuestra preferencia

- **ERROR DE LDCONFIG INICIAL**

> rm /lib/libts-0.0.so.0
> ln -s /lib/libts-0.0.so.0.1.1 /lib/libts-0.0.so.0

- **DESACTIVAR TTYS1 DE LA PANTALLA LCD**

- Abrir el fichero /etc/inittab y comentar la siguiente línea:
#1:2345:respawn:/sbin/getty 38400 tty1

- **DESACTIVAR MENSAJE AGING TEST Y LEDSACC**

- Abrir el fichero /etc/init.d/rc y comentar las siguientes 3 líneas:
#if [[-x /led_acc]]; then
echo " start aging test..."
/led_acc 1440 &
#fi

- **DESACTIVAR MENSAJE 'SYSTEM BOOTING: Please wait...'**

Abrir el fichero /etc/init.d/banner y comentar las siguientes líneas:

#echo > $vtmaster

#echo "Please wait: booting..." > $vtmaster

- **DESACTIVAR MENSAJE ERROR DEL CLOCK DEL SISTEMA**

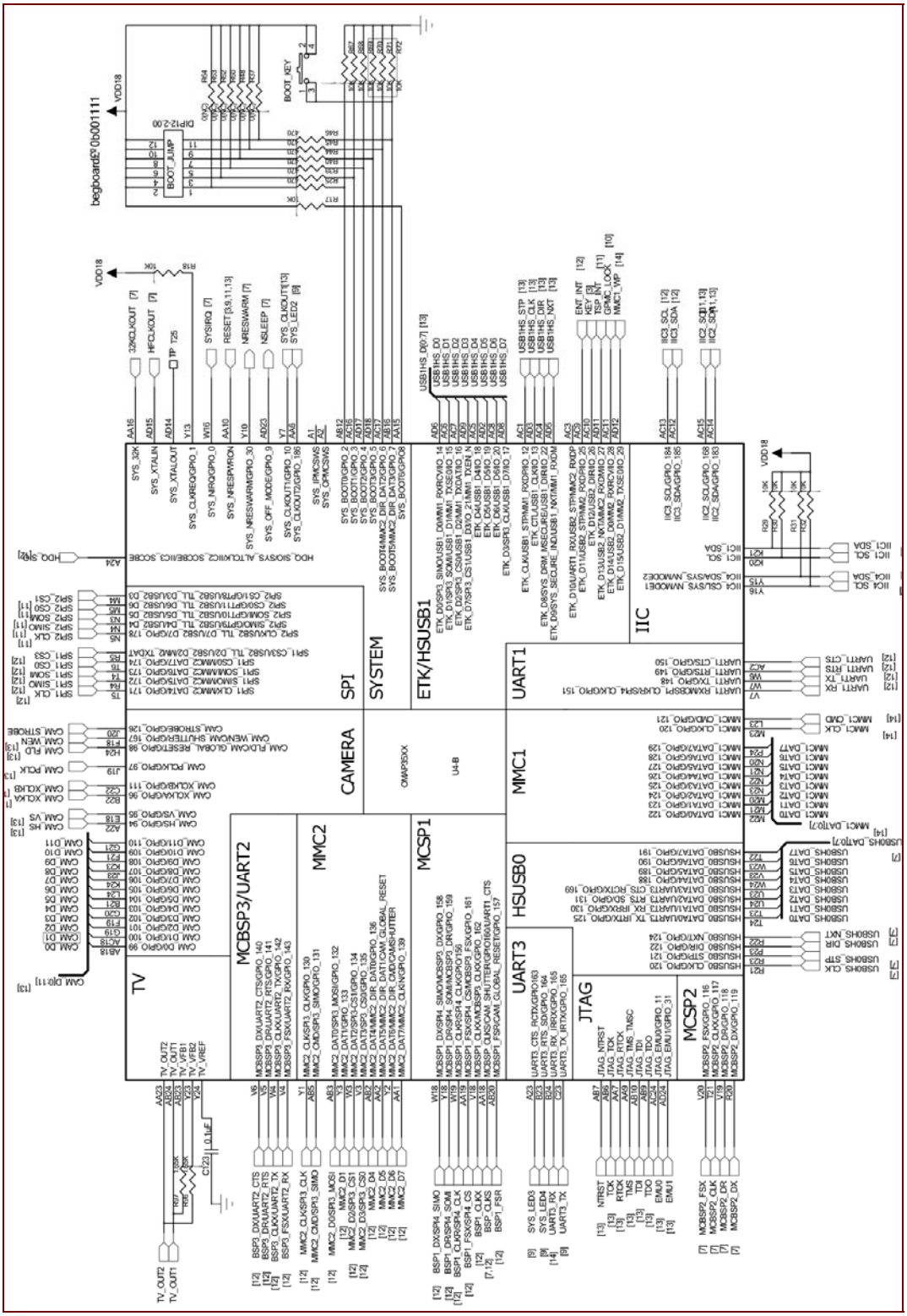
- Abrir el fichero /etc/init.d/bootmisc.sh y comentar la siguiente línea:
#/etc/init.d/hwclock.sh start

DESACTIVAR MENSAJES ERROR DE UDEVD

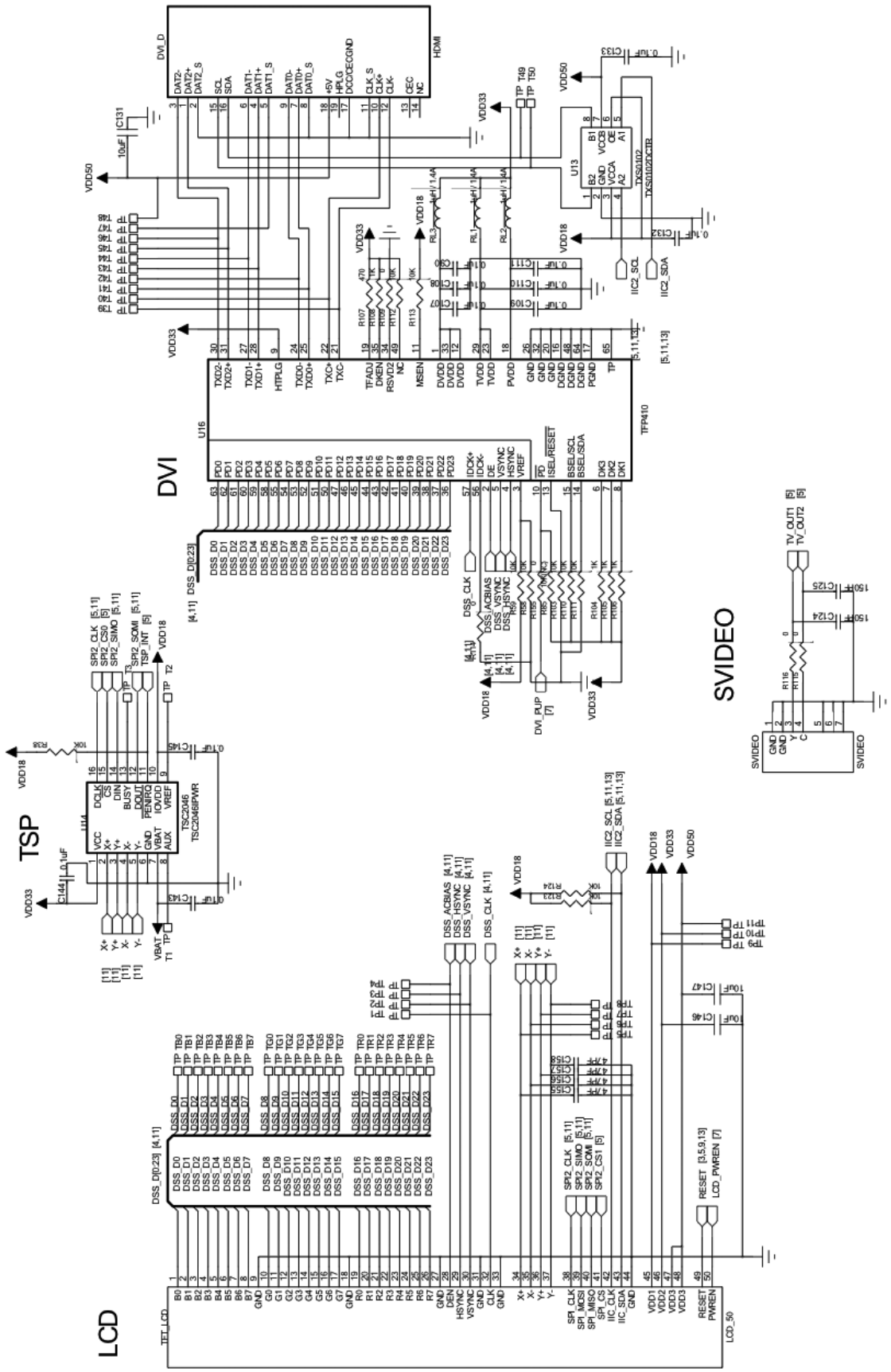
Abrir el fichero /etc/udev/rules.d/permissions.rules y comentar con # todas las líneas en las que aparezcan algunas de las siguientes palabras:

- scanner
- nvram
- tss
- fuse
- kvm
- rdma

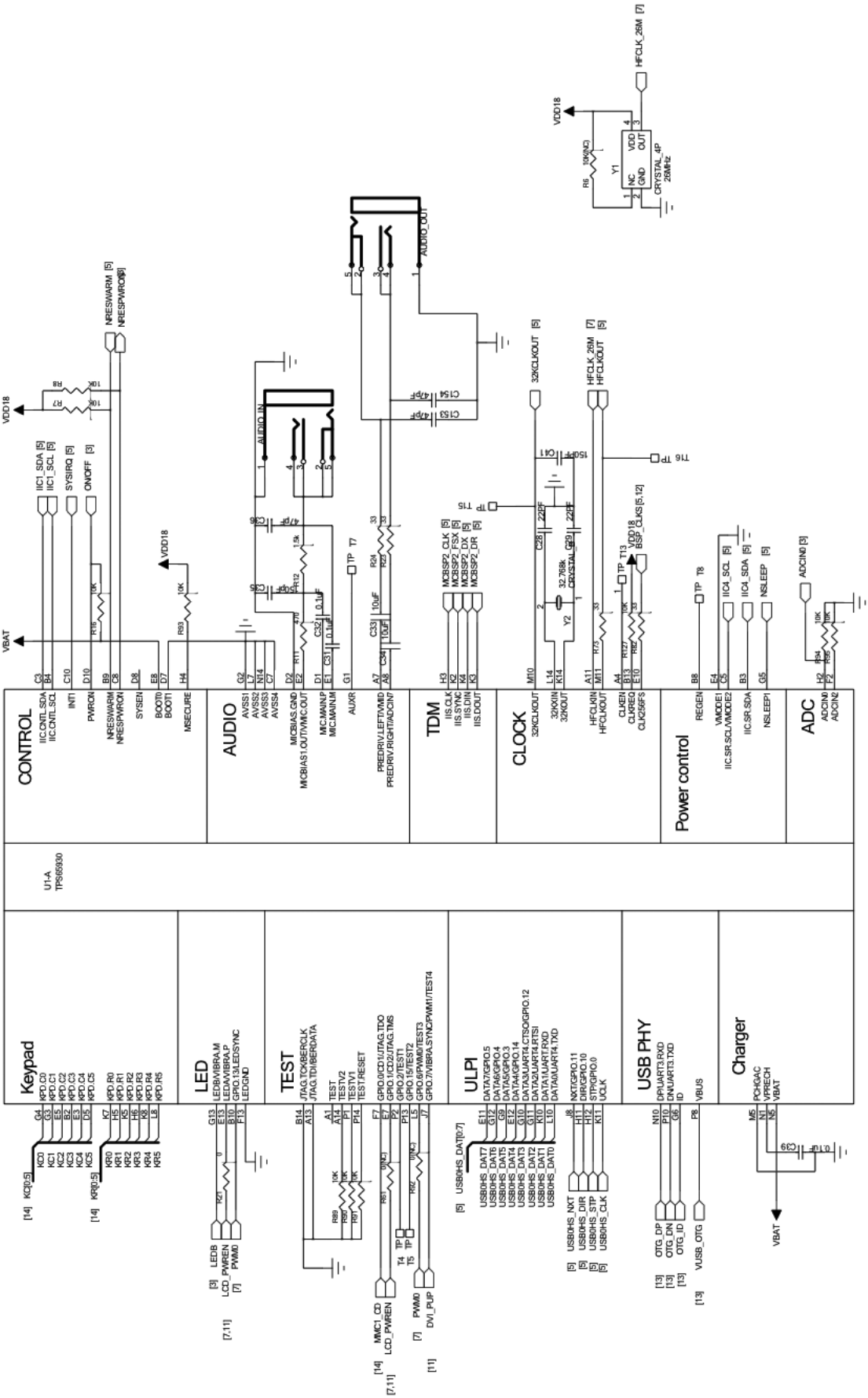
ANEXO II. ESQUEMAS ELÉCTRICOS



Conexiones principales del OMAP3530



Salida de vídeo a LCD, DVI y pantalla táctil



TPS65930: Audio, USB y Alimentación

ANEXO III. GUÍAS DE INSTALACIÓN

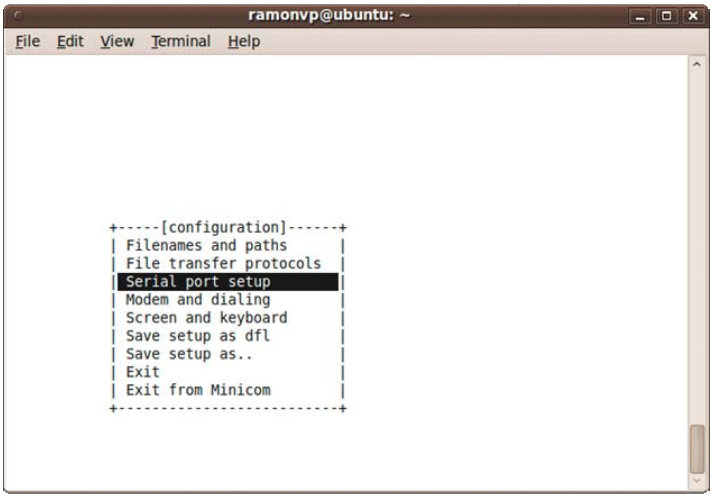
1. Minicom

Para iniciar la configuración de *minicom*, debemos de teclear lo siguiente en la línea de comandos de Linux:

> *> minicom -s*

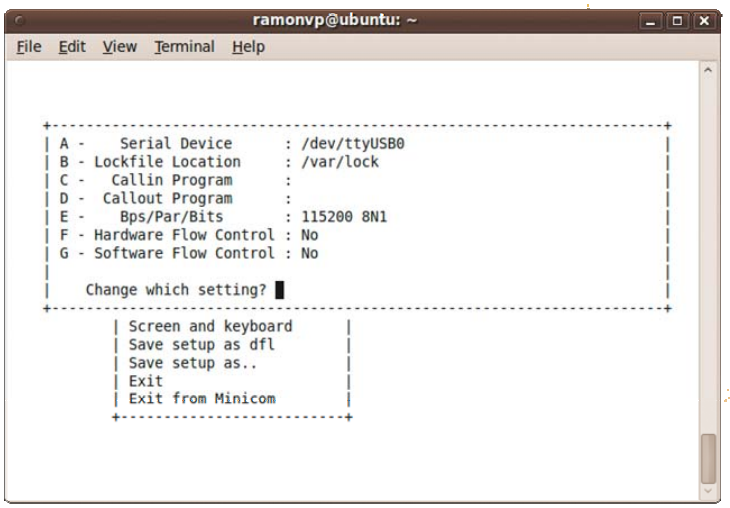
A continuación nos aparecerá la pantalla de bienvenida del *minicom*. En las imágenes siguientes, el texto aparece en inglés, puesto que el sistema operativo Ubuntu Linux que se instaló en el ordenador de desarrollo se hizo también en inglés. Las traducciones al castellano, no obstante, son bastante literales y simples de interpretar, con lo que no debería de haber ninguna confusión.

Este es el aspecto del menú principal del *minicom*:

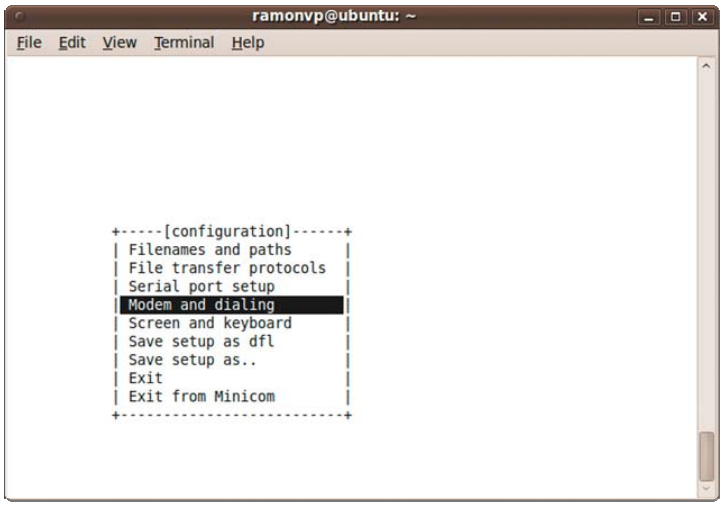


Elegimos la opción “Serial port setup” y pulsamos Enter.

Pulsamos la tecla “a” y escribiremos nuestro puerto serie. En el ejemplo mostrado aquí, /dev/ttyUSB0 es un conversor USB a puerto serie. Pulsamos Enter de nuevo cuando hayamos modificado el Serial Device. También hay que modificar la velocidad del puerto a 115200, 8 bits y paridad

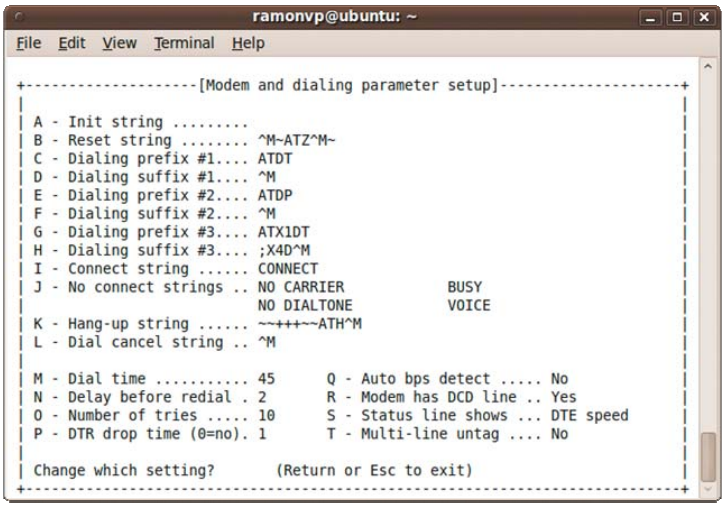


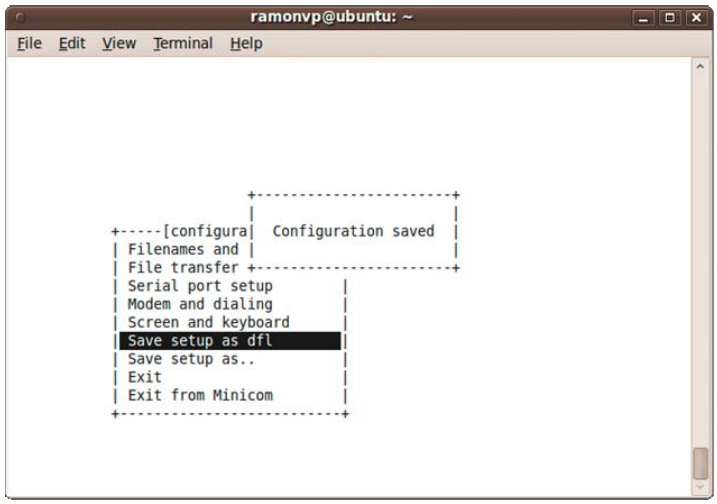
par. Esto se indica mediante "*115200 8N1"*. Asegurarse de que *Hardware Flow Control* está en *No* y *Software Flow Control* en *No*. Finalmente pulsamos *Enter* o *ESC* para volver al menú anterior.



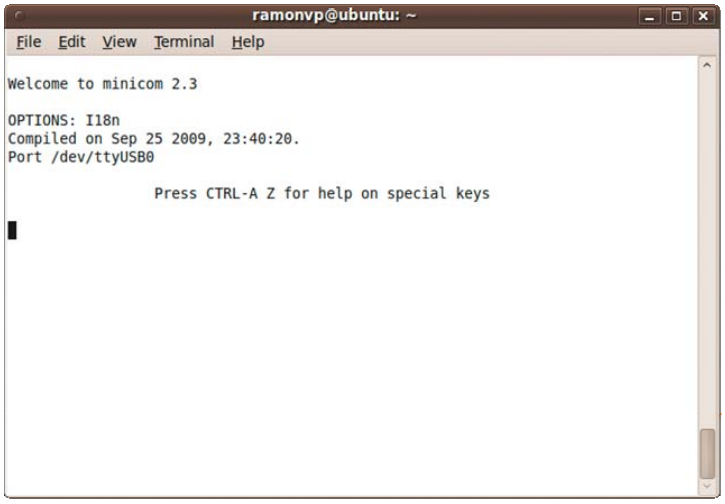
En el menú principal, entraremos en "Modem and dialing".

Aquí nos aseguraremos de que en la "*Init string*" esté vacío. Para ello pulsamos "a" y borramos cualquier comando que hubiera escrito anteriormente. Esto lo que evitará es enviar una secuencia de caracteres cada vez que abramos el puerto serie. Volvemos al menú principal de nuevo con *ENTER* o *ESC*.





Para no tener que hacer esta configuración cada vez que queramos conectar con la placa, lo guardaremos como configuración por defecto. Para ello pulsamos sobre "*Save setup as dfl*".

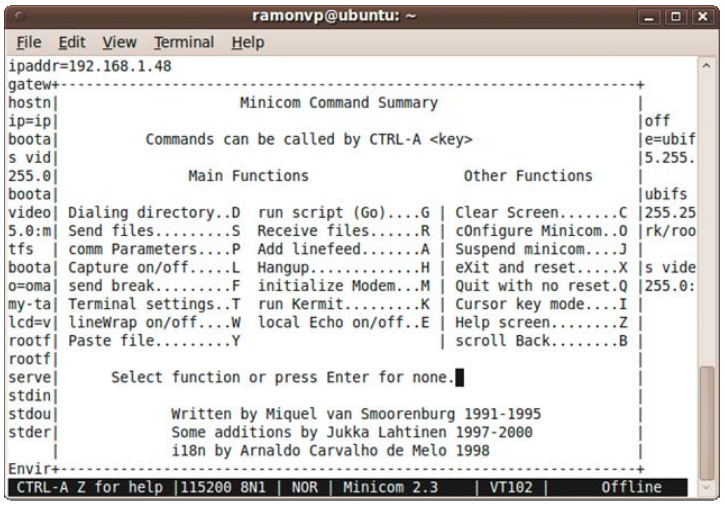


Una vez hecho esto, ya podemos abrir el puerto serie saliendo de la configuración, seleccionando la opción "*Exit*", lo que nos abrirá el puerto directamente.

La próxima vez que queramos abrir el puerto serie con *minicom*, únicamente habrá que ejecutar el programa llamándolo desde la línea de comandos:

```
> minicom
```

Para el caso concreto de la placa DevKit8000, y especialmente en la configuración de sus parámetros de arranque, es necesario desactivar del *minicom* la opción de rejuntado de líneas. Para ello, una vez abierto *minicom*, pulsaremos *CTRL+A*, y a continuación *Z*, para activar el menú de opciones.

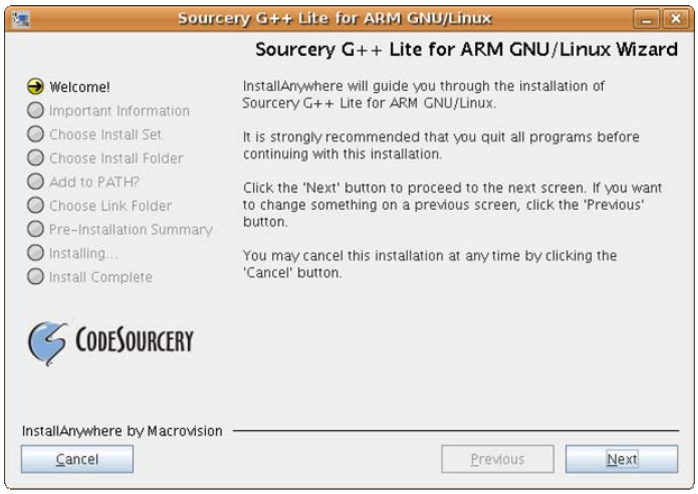


Pulsaremos la tecla "*w*" para activar el *lineWrap*, y veremos entonces en la pantalla el mensaje "*Linewrap ON*". Automáticamente esto ya nos devuelve al programa y podremos seguir usando el terminal.

2. CodeSourcery G++ GNU/Linux Toolchain

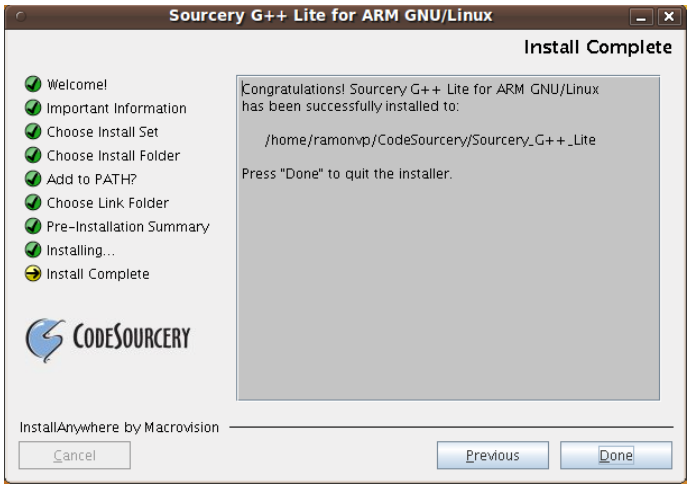
Para instalar el toolchain de Codesourcery, recordemos que hay que cambiar la shell por defecto de dash a bash, tal y como se explica en el capítulo 2. Una vez hecho esto, iniciamos el proceso de instalación ejecutando el fichero binario que acompaña al CD del fabricante o bien el fichero que se puede descargar de su página web en http://www.codesourcery.com.

```
>/bin/sh arm-2009q3-67-arm-none-linux-gnueabi.bin
```



El programa de instalación del *toolchain* consiste en un asistente gráfico que nos va guiando paso a paso, con lo que su instalación no resulta muy complicada. Iremos pinchando en el botón "*Next*" y aceptando las opciones por defecto.

Al finalizar la instalación, nos aparecerá una ventana como la siguiente. Pulsamos el botón "*Done*" y con esto la instalación ya habrá finalizado.



El instalador debería de haber añadido al PATH del sistema la ruta con las nuevas herramientas, de forma que podamos invocarlas desde cualquier directorio del sistema, igual que si llamásemos al gcc tradicional. Podemos comprobar esto escribiendo en un terminal lo siguiente:

```
> arm-none-linux-gnueabi-gcc --version
```

Si todo está bien, debería aparecer la versión del compilador en el terminal, algo similar a:

```
arm-none-linux-gnueabi-gcc (Sourcery G++ Lite 2009q3-67) 4.4.1
Copyright (C) 2009 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There
is NO warranty; not even for MERCHANTABILITY or FITNESS FOR A
PARTICULAR PURPOSE.
```

Si fallase, hay que añadirlo a mano en *$HOME/.bashrc*. Abriremos dicho fichero con cualquier editor de texto y añadiremos la siguiente línea:

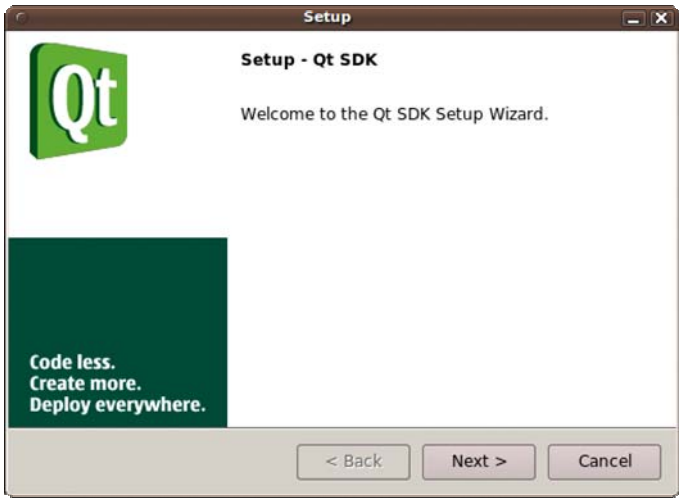
```
export
PATH=$HOME/CodeSourcery/Sourcery_G++_Lite/bin:$HOME/tools:$PATH
```

También aprovecharemos y declararemos una variable de entorno con el PATH a nuestra *toolchain*, ya que nos resultará útil para más adelante cuando compilemos programas y librerías:

```
export TOOLCHAIN=$HOME/CodeSourcery/Sourcery_G++_Lite/arm-none-
linux-gnueabi
```

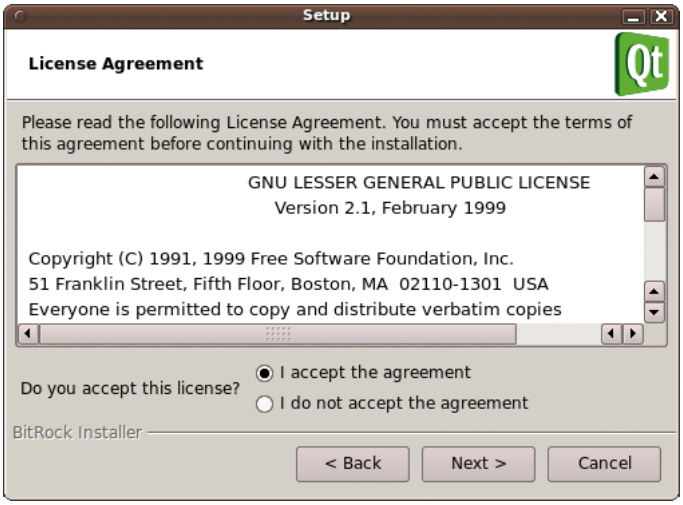
En el CD del fabricante de la placa vienen unas herramientas adicionales que necesitaremos a la hora de compilar nuevos kernels y cuando queramos actualizar el sistema de ficheros rootfs presente en la memoria flash en la placa de desarrollo. Es por esto que también se ha añadido al PATH del sistema para evitar así futuros errores de muchos scripts de configuración, que presuponen que estas herramientas son visibles en todo el árbol de directorios. No hay que descuidarse de copiar estas herramientas en *$HOME/tools*. Las podremos encontrar en el directorio */tools* del CD.

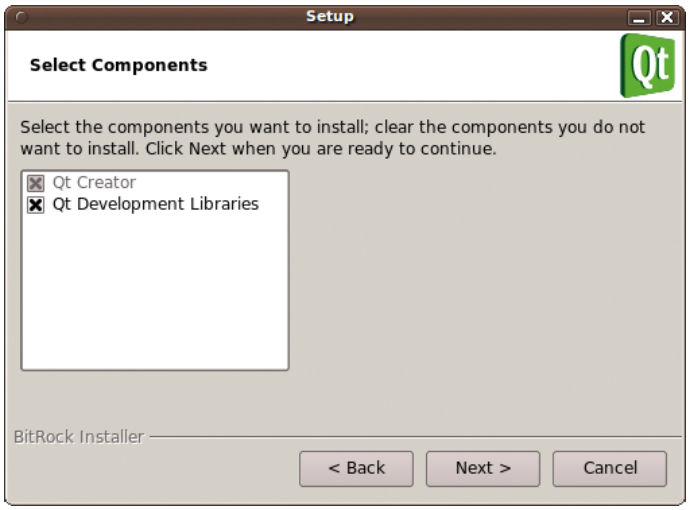
3. QtCreator de Nokia



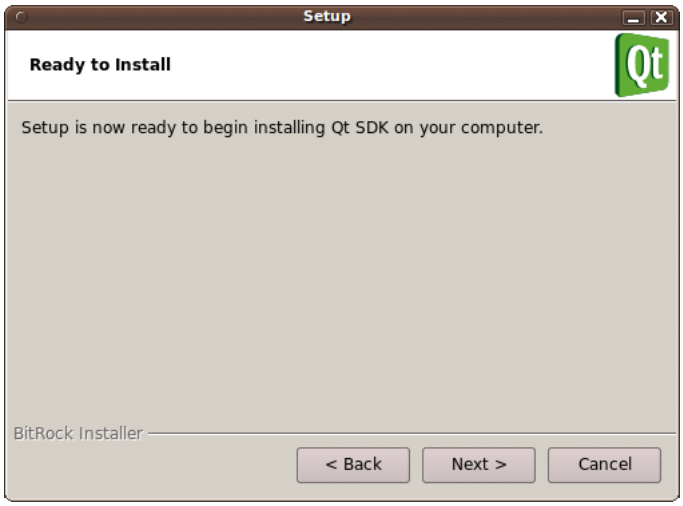
Al iniciar la instalación, nos aparece la pantalla de bienvenida del instalador. Pinchamos en el botón “Next”.

Para poder seguir con la instalación, es necesario aceptar las condiciones que nos imponen en el contrato de licencia. Marcamos la opción conforme se acepta el acuerdo y pinchamos en “Next”.



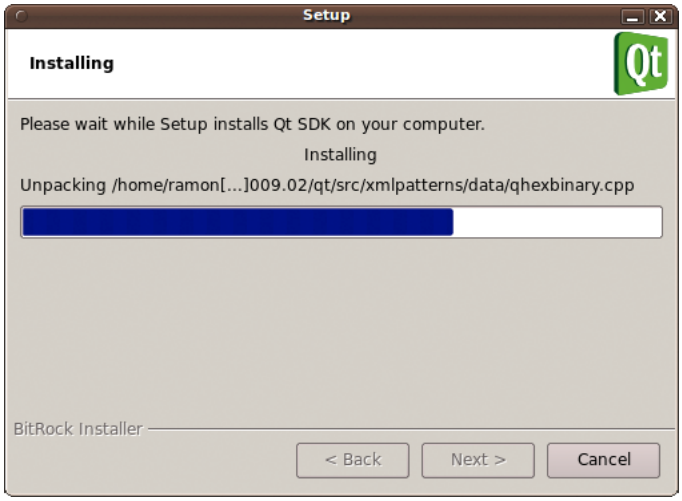


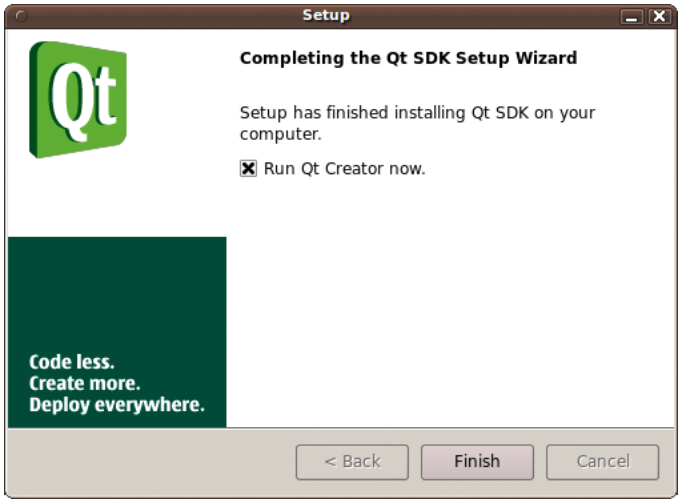
A continuación nos aparece una lista con las opciones que podemos instalar. En principio debemos marcar todas las opciones, dado que se presupone que no tenemos ningún componente instalado con anterioridad.



El programa ya está listo para empezar a instalar Qt Creador. Para iniciar el proceso, pinchamos en “Next”.

Esperamos a que finalice el proceso de instalación. Dicho proceso puede llevar cierto tiempo, incluso en ordenador moderno y potento. Veremos como la barra de progreso irá rellenándose conforme vaya faltando menos para terminar.





Cuando el proceso haya finalizado, veremos esta pantalla preguntando si queremos abrir Qt Creator al pinchar en el botón “Finish”. Podemos abrir ahora el programa si lo deseamos.

Tras la instalación, si no hemos marcado la opción de abrir el programa, siempre podemos abrirlo a través del menú de Ubuntu Linux, en Aplicaciones, en la sección de Programación.

Finalmente se nos abrirá una pantalla de bienvenida similar a la siguiente:

Ya tenemos listo el programa para empezar a trabajar. En la zona izquierda vemos las siguientes secciones:

- **Welcome**: sección de bienvenida, donde permite abrir proyectos recientes
- **Edit**: donde se edita el código fuente de los ficheros del proyecto
- **Debug**: se utiliza para depurar la aplicación, línea a línea
- **Projects**: desde aquí gestionaremos los parámetros de configuración de nuestros proyectos
- **Help**: aquí encontraremos toda la documentación de ayuda relativa al uso del entorno Qt Creator.
- **Output**: muestra en grande la salida generada en las acciones del programa

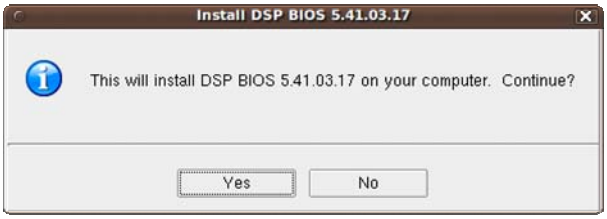
4. DSP/BIOS LINK

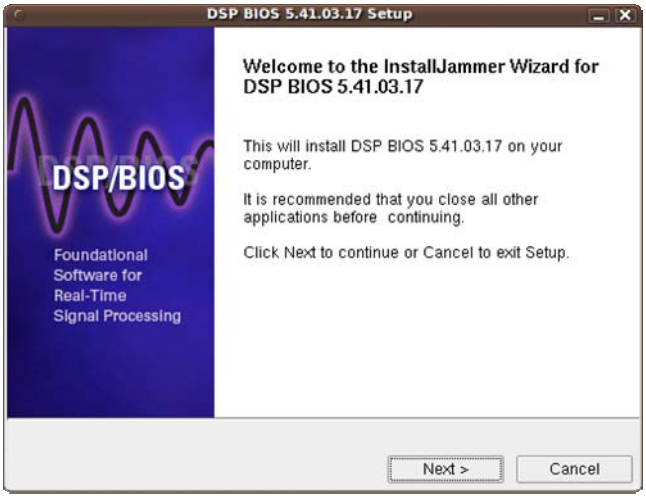
Empezamos instalando el DSP BIOS:

```
> ./bios_setuplinux_5_41_03_17.bin
```

Elegimos el idioma de nuestra preferencia, en esta guía mantendremos el inglés.

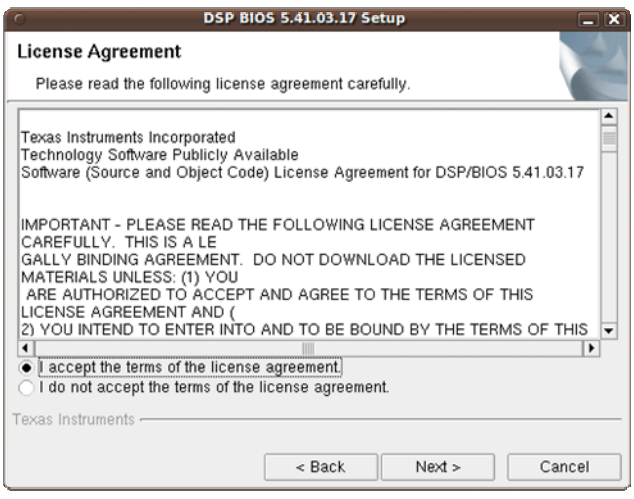
Indicamos que SI para seguir cargando el instalador.

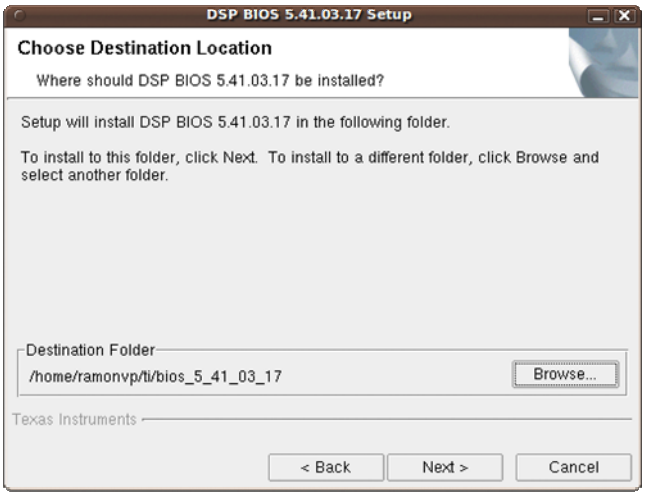




En la pantalla de bienvenida del instalador, pulsaremos el botón “Next”.

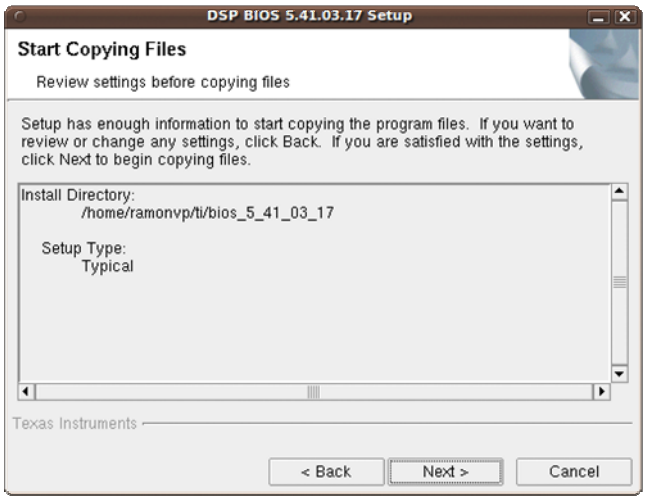
Como suele ser habitual en herramientas de desarrollo gratuitas, se nos pide que aceptemos un acuerdo de licencia, para lo cual marcamos la opción “I accept”. Seguimos pulsando el botón “Next”.

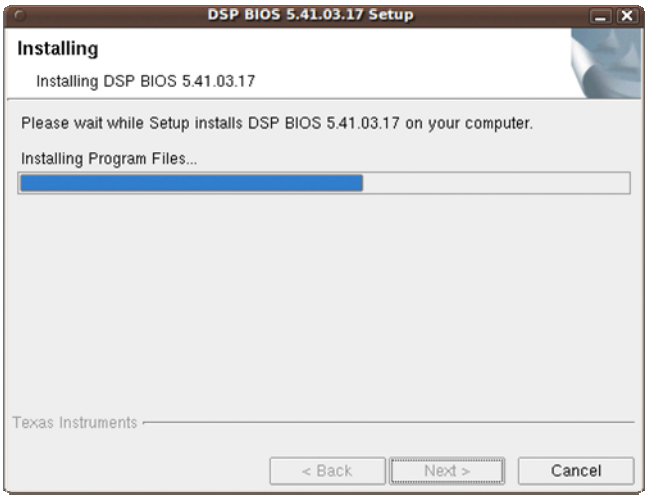




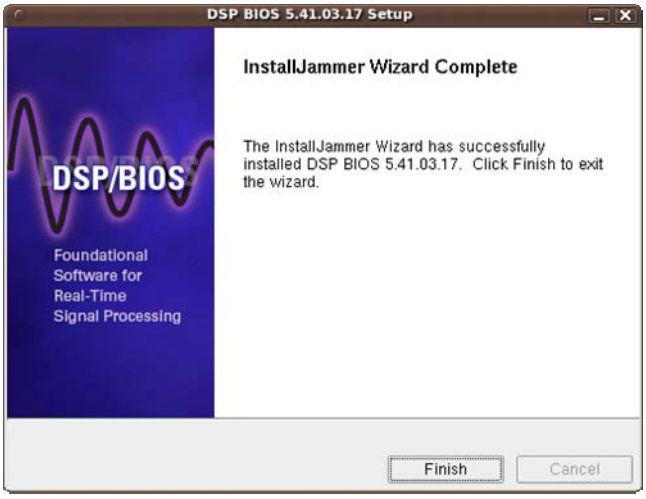
En la siguiente pantalla nos preguntan en qué directorio deseamos instalar los ficheros. Dejaremos la opción por defecto que aparezca, pero nos aseguraremos de que el directorio base sea *$HOME/TI*.

Una vez ya esté todo listo, iniciamos el proceso de instalación pulsando sobre el botón “Next”.





Veremos cómo el proceso de copia de ficheros va progresando poco a poco.



Cuando el proceso haya finalizado, nos aparecerá la pantalla de fin de instalación. Pulsamos “Finish” y daremos la instalación por concluida.

A continuación instalaremos las herramientas restantes. Dado que los instaladores son prácticamente iguales que el anterior, se omiten las imágenes relativas a su instalación. Únicamente recordar que es conveniente que todas las herramientas finalmente queden instaladas bajo el directorio *$HOME/TI*.

Para iniciar los instaladores de las otras dos herramientas restantes, escribiremos:

```
> ./xdctools_setuplinux_3_16_01_27.bin
```

```
> ./ti_cgt_c6000_6.1.13_setup_linux_x86
```

ANEXO IV. COMPILACIÓN CRUZADA

Normalmente encontraremos los programas y librerías en código fuente, con lo que será necesario compilar el paquete para poder ejecutarlo en nuestra máquina. Como lo que deseamos es que dicha aplicación o librería corra sobre la plataforma de la placa DevKit8000, lo que deberemos hacer es una compilación cruzada, utilizando lo que en inglés se denomina un cross-compiler o compilador cruzado.

Para facilitar la explicación de cómo se realiza este proceso, se ha comentado en este anexo un ejemplo, que consiste en lograr obtener un fichero binario ejecutable para la placa DevKit8000 y para Embedded Linux: el conocido editor de ficheros en modo texto *Nano*.

Observaciones generales

A la hora de compilar veremos que la mayoría de paquetes y de librerías tienen un sistema de compilación muy parecido entre ellos. Concretamente, la herramienta por excelencia para compilar en entornos Unix es la utilidad *make*. Esta utilidad tiene su propia sintaxis, que no explicaremos aquí por lo extenso del tema, pero sí que vamos a comentar las opciones en la línea de comandos que son necesarias para una compilación cruzada.

El primer paso suele ser ejecutar un script de configuración denominado *./configure* . La misión de este script es la de crear los ficheros *Makefile* que más tarde usará *make* para realizar el proceso de compilación. Este script es el importante de cara a configurar correctamente entornos de compilación cruzada, porque es aquí donde hay que poner los parámetros adecuados. Los más habituales son:

--build=i686-pc-linux-gnu: esta opción indica que la compilación se realizará sobre esta plataforma. i686 es la plataforma estándar de un ordenador moderno actual, por ejemplo, basado en un Intel Dual Core. Muchas veces se podrá omitir, porque el propio script ya averigua esta información por su cuenta.

--host=arm-none-linux-gnueabi: La opción *host* indica a la utilidad de compilación qué tipo de plataforma va ser quién ejecute esta aplicación. En este caso, le estamos indicando que queremos compilar para plataforma ARM. Es por ello que automáticamente el compilador añadirá como prefijo lo indicado a las utilidades habituales de compilación, quedando por ejemplo:

gcc → arm-none-linux-gnueabi-gcc
ld → arm-none-linux-gnueabi-ld
etc…

Esto modifica las variables de entorno típicas empleadas en programación, como por ejemplo CC y LD.

--prefix $path/salida/deseado: esta opción indica un directorio en el cual deseamos que se depositen todos los ficheros que se compilen cuando llamemos a la orden *make install*, que lo que hará es copiar los ficheros donde originalmente quedaron compilados a este destino final. Normalmente suele tener un valor predeterminado que se debe consultar en cada caso, como por ejemplo /usr/bin/nano. Como no es conveniente que los binarios de una plataforma se mezclen con los binarios de la plataforma del ordenador host, esta opción resulta indispensable.

Vistos estos detalles, pasamos al ejemplo. Empezaremos por descargar y compilar la librería ncurses, que es la que permite ejecutar en una ventana terminal programas que necesitan posicionar el cursor y mostrar colores.

LIBRERIA NCURSES PARA CONSOLA EN TEXTO

Descargar de: http://ftp.gnu.org/pub/gnu/ncurses/ncurses-5.7.tar.gz (2.3 MB)

```
> cd $HOME/projects
> tar xvf ncurses-5.7.tar.gz
> mkdir $HOME/projects/nano_compiled
> cd ncurses-5.7
> ./configure --build=i686-pc-linux-gnu --host=arm-none-linux-
gnueabi --prefix $HOME/projects/nano_compiled
> make
> make install
```

Tras esto ya tendremos en el directorio $HOME/projects/nano_compiled la librería ncurses compilada. La dejaremos en este directorio para que a continuación la compilación del paquete nano sepa encontrarla.

UTILIDAD NANO PARA EDITAR FICHEROS

Descargar de: http://www.nano-editor.org/dist/v2.2/nano-2.2.3.tar.gz (1.5 MB)

```
> ./configure --build=i686-pc-linux-gnu --host=arm-none-linux-
gnueabi --prefix $HOME/projects/nano_compiled
> make
> make install
```

Luego copiaremos el contenido de $HOME/projects/nano_compiled a los directorios correspondientes en la placa DevKit8000, que podrían ser /bin o /usr/bin, por ejemplo.

No hay que olvidar que en la placa hay que definir 2 variables de entorno que necesita *nano* para poder funcionar. Una representa el directorio donde se encuentra toda la información de configuración de los distintos terminales. La otra variable representa el tipo de terminal que deseamos emular. Para la conexión a través de puerto serie resulta indispensable la emulación del terminal tipo VT100:

```
export TERMINFO=/usr/share/terminfo
export TERM=vt100
```

Para guardar permanentemente estos cambios y no tener que reescribir estas variables de entornos cada vez, lo podemos hacer en el fichero /etc/profile de la placa.

PÁGINAS DE MANUALES

Ser recomienda la instalación de los manuales en el sistema host que empleemos de desarrollo, puesto que resulta ser una fuente inestimable de información y de ayuda en el desarrollo de la aplicaciones y configurando sistemas Linux. Para ello, se deberán instalar los siguientes paquetes con la utilidad apt-get: manpages, manpages-dev, manpages-posix, manpages-posix-dev, freebsd-manpages.

ANEXO V. FUNCIONES INCORPORADAS EN DSPLIB

Table 3–2. Adaptive Filtering

Functions	Description	Page
long DSP_firlms2(short *h, short *x, short b, int nh)	LMS FIR	4-2

Table 3–3. Correlation

Functions	Description	Page
void DSP_autocor(short *r,short *x, int nx, int nr)	Autocorrelation	4-4

Table 3–4. FFT

Functions	Description	Page
void DSP_fft16x16(short *w, int nx, short *x, short *y)	Complex out of place, Forward FFT mixed radix with digit reversal. Input/Output data in Re/Im order.	4-8
void DSP_fft16x16_imre(short *w, int nx, short *x, short *y)	Complex out of place, Forward FFT mixed radix with digit reversal. Input/Output data in Im/Re order.	4-11
void DSP_fft16x16r(int nx, short *x, short *w, unsigned char *brev, short *y, int offset, int n_max)	Mixed radix FFT with scaling and rounding, digit reversal, out of place. Input and output: 16 bits, Twiddle factor: 16 bits.	4-12
void DSP_fft16x32(short *w, int nx, int *x, int *y)	Extended precision, mixed radix FFT, rounding, digit reversal, out of place. Input and output: 32 bits, Twiddle factor: 16 bits.	4-22
void DSP_fft32x32(int *w, int nx, int *x, int *y)	Extended precision, mixed radix FFT, rounding, digit reversal, out of place. Input and output: 32 bits, Twiddle factor: 32 bits.	4-24
void DSP_fft32x32s(int *w, int nx, int *x, int *y)	Extended precision, mixed radix FFT, digit reversal, out of place., with scaling and rounding. Input and output: 32 bits, Twiddle factor: 32 bits.	4-26

Tabla 1: Filtrado Adaptativo, Correlación y FFT

Functions	Description	Page
void DSP_ifft16x16(short *w, int nx, short *x, short *y)	Complex out of place, Inverse FFT mixed radix with digit reversal. Input/Output data in Re/Im order.	4-26
void DSP_ifft16x16_imre(short *w, int nx, short *x, short *y)	Complex out of place, Inverse FFT mixed radix with digit reversal. Input/Output data in Re/Im order.	4-26
void DSP_ifft16x32(short *w, int nx, int *x, int *y)	Extended precision, mixed radix IFFT, rounding, digit reversal, out of place. Input and output: 32 bits, Twiddle factor: 16 bits.	4-32
void DSP_ifft32x32(int *w, int nx, int *x, int *y)	Extended precision, mixed radix IFFT, digit reversal, out of place, with scaling and rounding. Input and output: 32 bits, Twiddle factor: 32 bits.	4-34

Table 3–5. Filtering and Convolution

Functions	Description	Page
void DSP_fir_cplx(short *x, short *h, short *r, int nh, int nr)	Complex FIR Filter (nh is a multiple of 2)	4-36
void DSP_fir_cplx_hM4X4(short *x, short *h, short *r, int nh, int nr)	Complex FIR Filter (nh is a multiple of 4)	4-36
void DSP_fir_gen (short *x, short *h, short *r, int nh, int nr)	FIR Filter (any nh)	4-40
void DSP_fir_gen_hM17_rA8X8 (short *x, short *h, short *r, int nh, int nr)	FIR Filter (r[] must be double word aligned, nr must be multiple of 8)	4-40
void DSP_fir_r4 (short *x, short *h, short *r, int nh, int nr)	FIR Filter (nh is a multiple of 4)	4-44
void DSP_fir_r8 (short *x, short *h, short *r, int nh, int nr)	FIR Filter (nh is a multiple of 8)	4-48
void DSP_fir_r8_hM16_rM8A8X8 (short *x, short *h, short *r, int nh, int nr)	FIR Filter (r[] must be double word aligned, nr is a multiple of 8)	4-48
void DSP_fir_sym (short *x, short *h, short *r, int nh, int nr, int s)	Symmetric FIR Filter (nh is a multiple of 8)	4-50

Tabla 2: FFT (continuación), filtrado y convolución

Table 3–5. Filtering and Convolution (Continued)

Functions	Description	Page
void DSP_iir(short Input, short *Coefs, int nCoefs, short State)	IIR Filter	4-52
void DSP_iir_lat(short *x, int nx, short *k, int nk, int *b, short *r)	All–pole IIR Lattice Filter	4-54

Table 3–6. Math

Functions	Description	Page
int DSP_dotp_sqr(int G, short *x, short *y, int *r, int nx)	Vector Dot Product and Square	4-56
int DSP_dotprod(short *x, short *y, int nx)	Vector Dot Product	4-58
short DSP_maxval (short *x, int nx)	Maximum Value of a Vector	4-60
int DSP_maxidx (short *x, int nx)	Index of the Maximum Element of a Vector	4-61
short DSP_minval (short *x, int nx)	Minimum Value of a Vector	4-63
void DSP_mul32(int *x, int *y, int *r, short nx)	32-bit Vector Multiply	4-64
void DSP_neg32(int *x, int *r, short nx)	32-bit Vector Negate	4-66
void DSP_recip16 (short *x, short *rfrac, short *rexp, short nx)	16-bit Reciprocal	4-67
int DSP_vecsumsq (short *x, int nx)	Sum of Squares	4-69
void DSP_w_vec(short *x, short *y, short m, short *r, short nr)	Weighted Vector Sum	4-70

Table 3–7. Matrix

Functions	Description	Page
void DSP_mat_mul(short *x, int r1, int c1, short *y, int c2, short *r, int qs)	Matrix Multiplication	4-71
void DSP_mat_trans(short *x, short rows, short columns, short *r)	Matrix Transpose	4-73

Tabla 3: Filtrado y convolución (continuado), funciones matemáticas y matriciales

Table 3–8. Miscellaneous

Functions	Description	Page
short DSP_bexp(int *x, short nx)	Max Exponent of a Vector (for scaling)	4-74
void DSP_blk_eswap16(void *x, void *r, int nx)	Endian-swap a block of 16-bit values	4-76
void DSP_blk_eswap32(void *x, void *r, int nx)	Endian-swap a block of 32-bit values	4-78
void DSP_blk_eswap64(void *x, void *r, int nx)	Endian-swap a block of 64-bit values	4-80
void DSP_blk_move(short *x, short *r, int nx)	Move a Block of Memory	4-82
void DSP_fltoq15 (float *x,short *r, short nx)	Float to Q15 Conversion	4-83
int DSP_minerror (short *GSP0_TABLE,short *errCoefs, int *savePtr_ret)	Minimum Energy Error Search	4-85
void DSP_q15tofl (short *x, float *r, short nx)	Q15 to Float Conversion	4-87

Tabla 4: Funciones varias